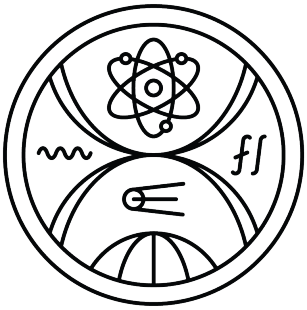
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS

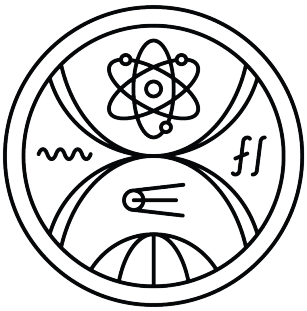


KUBERNETES SECURITY ASSESSMENT

Master thesis

2024 Bc. Pavel Semenov

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS



KUBERNETES SECURITY ASSESSMENT

Master thesis

Study program:	Applied informatics
Branch of study:	Applied informatics
Department:	Department of Applied Informatics
Supervisor:	prof. RNDr. Richard Ostertág, PhD.
Consultant:	Mgr. Ľubomír Firment

Bratislava, 2024 Bc. Pavel Semenov

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Bc. Pavel Semenov
Study programme: Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study: Computer Science
Type of Thesis: Diploma Thesis
Language of Thesis: English
Secondary language: Slovak

Title: Kubernetes security assessment

Annotation: Kubernetes has been gaining popularity rapidly in recent years as more and more enterprise solutions are subjected to cloud transformation and more companies are looking for the ways to increase development efficiency and reduce development costs. This brings new concerns from clients and stakeholders about the security of Kubernetes and its exposure to cyber-attacks.

Aim:

This thesis studies, compares and evaluates the state-of-the-art tools designed to discover vulnerabilities concerning the cluster configuration, running pods or cluster itself. Assessment is carried out in both local cluster setup predisposed with multiple vulnerabilities and real-world enterprise cloud infrastructure. Based on the assessment results we intend either to improve one of the existing tools or develop a Kubernetes security framework of our own, which will be able to provide better results in addressing the cluster security.

Literature:

V. B. Mahajan and S. B. Mane, "Detection, Analysis and Countermeasures for Container based Misconfiguration using Docker and Kubernetes", 2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS), 2022, pp. 1-6, doi: 10.1109/IC3SIS54991.2022.9885293. https://ieeexplore.ieee.org/document/9885293

D. B. Bose, A. Rahman and S. I. Shamim, "'Under-reported' Security Defects in Kubernetes Manifests", 2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS), 2021, pp. 9-12, doi: 10.1109/EnCyCriS52570.2021.00009. https://ieeexplore.ieee.org/document/9476056

Castillo Rivas, D.A., Guamán, D. (2021). "Performance and Security Evaluation in Microservices Architecture Using Open Source Containers". In: Botto-Tobar, M., Montes León, S., Camacho, O., Chávez, D., Torres-Carrión, P., Zambrano Vizuete, M. (eds) Applied Technologies. ICAT 2020. Communications in Computer and Information Science, vol 1388. Springer, Cham. https://doi.org/10.1007/978-3-030-71503-8_37

Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi (2022). "Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster". In Proceedings of the 17th International Conference on Availability, Reliability and Security (ARES '22). Association for Computing Machinery, New York, NY, USA, Article 117, 1–9. https://doi.org/10.1145/3538969.3543810

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Computing Machinery, New York, NY, USA, Article 117, 1–9. https://doi.org/10.1145/3538969.3543810

Vedúci: RNDr. Richard Ostertág, PhD.
Konzultant: Mgr. Ľubomír Firment
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 07.12.2022

Dátum schválenia: 07.12.2022 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

I hereby declare that I have written this thesis by myself, only with help of referenced literature, under the careful supervision of my thesis advisor.

.....

Bratislava, 2024 Bc. Pavel Semenov

Acknowledgement

First, I would like to express my gratitude to Mgr. Ľubomír Firment for his guidance during the whole thesis and invaluable expertise in Kubernetes that made this thesis possible. I'd also like to thank my supervisor RNDr. Richard Ostertág, PhD. for his insightful feedback.

Abstract

Kubernetes has been gaining popularity rapidly in recent years as more and more enterprise solutions are subjected to cloud transformation and more companies are looking for the ways to increase development efficiency and reduce development costs. This brings new concerns from clients and stakeholders about the security of Kubernetes and its exposure to cyber-attacks. This thesis studies, compares and evaluates the state-of-the-art tools designed to discover vulnerabilities concerning the cluster configuration, running pods or cluster itself. Assessment is carried out in both local cluster setup predisposed with multiple vulnerabilities and real-world enterprise cloud infrastructure. Based on the assessment results we intend either to improve one of the existing tools or develop a Kubernetes security framework of our own, which will be able to provide better results in addressing the cluster security.

Keywords: kubernetes, security, test, cloud

Abstrakt

Kubernetes v posledných rokoch rýchlo získava na popularite, pretože čoraz viac spoločností hľadá spôsoby, ako zvýšiť efektivitu vývoja a znížiť náklady na vývoj. Táto zvýšená popularita so sebou prináša väčšie vystavenie kybernetickým útokom a zvýšené obavy zainteresovaných strán o bezpečnosť Kubernetes. Cieľom práce je porovnať a zhodnotiť moderné nástroje určené na odhaľovanie zraniteľností týkajúcich sa konfigurácie klastra, bežiacich podov alebo aj samotného klastra. Posúdenie bude prebiehať na lokálnom klastri s prednasadenými viacerými zraniteľnosťami, ako aj v reálnej podnikovej cloudovej infraštruktúre. Na základe výsledkov hodnotenia máme v úmysle buď vylepšiť niektorý z existujúcich nástrojov, alebo vyvinúť vlastný bezpečnostný rámec pre Kubernetes, ktorý bude schopný poskytnúť lepšie výsledky pri riešení klastrovej bezpečnosti.

Kľúčové slová: kubernetes, bezpečnosť, testovanie, cloud

Contents

List of Figures

List of Tables

List of Listings

Terminology

Terms

- **CI/CD pipeline**
  CI/CD pipeline is a set of automatic tasks to be executed upon specific action resulting in ether failure on some of the pipeline stages or successful application build and deployment on the target environment. The aforementioned action might be a push into the source code repository or a manual pipeline run request. CI/CD pipeline usually includes build, test and deploy stages.
- **Cloud**
  Term "cloud" is ususally used to describe an array of (remote, on-premise or hybrid) servers that operate as a single ecosystem used for various hosting services.
- **Cloud provider**
  A cloud provider is a company that offers cloud computing services, which include resources like storage, processing power, networking, databases, and software, delivered over the internet.
- **Vulnerability**
  When we say "vulnerability" in this paper, we mostly refer to the exploitable weakness inside the container. Depending on the context, it can also mean security threat, in general sense.
- **Misconfiguration**
  "Misconfiguration" refers explicitly to the incorrectly configured (or unconfigured) Kubernetes resource or component. "Misconfiguration" can be ontained by the Kubernetes operator configuring something wrong, not configuring something or configuring something incompletely.

Abbreviations

- **AI** – Artificial Intelligence.
- **AICPS** – American Institute of Certified Public Accountants.

- **API** – Application Programming Interface.
- **AWS** – Amazon Web Services.
- **CI/CD** – Continuous Integration/Continuous Deployment.
- **CIS** – Center of Internet Security.
- **CISA** – Cybersecurity and Infrastructure Security Agency.
- **CLI** – Command-line Interface.
- **CNCF** – Cloud Native Computing Foundation.
- **CPU** – Central Processing Unit.
- **CSI** – Container Storage Interface.
- **CSRF** – Cross Site Request Forgery.
- **CSV** – Comma-separated Values.
- **CVE** – Common Vulnerabilities and Exposures.
- **ECR** – Elastic Container Registry.
- **EU** – European Union.
- **ePHI** – Electronic Protected Health Information.
- **GCP** – Google Cloud Platform.
- **GDPR** – General Data Protection Regulation.
- **GUI** – Graphical User Interface.
- **HIPAA** – Health Insurance Portability and Accountability Act.
- **HTTP** – Hypertext Transfer Protocol.
- **IBM** – International Business Machines.
- **IaC** – Infrastructure as Code.
- **iSCSI** – Internet Small Computer Systems Interface.
- **K8s** – Kubernetes.
- **KSA** – Kubernetes Security Assessment.
- **NFS** – Network File System.
- **NIST** – National Institute of Standards and Technology.
- **NSA** – National Security Agency.
- **OS** – Operating System.

- **OWASP** – The Open Worldwide Application Security Project.
- **PCI-DSS** – Payment Card Industry Data Security Standard.
- **PDF** – Portable Document Format.
- **PVC** – PersistentVolumeClaim.
- **RBAC** – Role Based Access Control.
- **SBOM** – Software Bill of Materials.
- **SOC** – Service Organization Control.
- **TCP** – Transmission Control Protocol.
- **TLS** – Transport Layer Security.
- **VM** – Virtual Machine.
- **XSS** – Cross Site Scripting.

Motivation

As the world's biggest corporations start grasping the power of the cloud computing, stakeholders are raising concerns regarding the security of the most popular and accessible Container Orchestration platform - Kubernetes. Opinion of the experts on this matter varies significantly and this paper aims to make a contribution to this dispute by determining how well can be Kubernetes cluster's security monitored.

This paper is highly inspired and motivated by the author's own experience on multiple projects as DevSecOps consultant at IBM. As IBM desires a leading position in enterprise consulting world, it is also looking to modernize the clients' infrastructures by installing Kubernetes servers. IBM always wants to ensure customers of the safety of their data in the cloud, which makes this research highly valuable in contract negotiations.

We compare and evaluate the capabilities of the most popular security tools designed specifically for Kubernetes against official and unofficial Kubernetes security recommendations. Furthermore, we develop a Kubernetes monitoring tool that aims to compliment existing infrastructure with additional security monitoring capabilities. Our dashboard gathers scanner data, parses the results and list found misconfigurations and vulnerabilities in readable format.

At the moment of witing this paper, a number of academic papers on Kubernetes security is limited and the topics of research have little in common with our paper. For instance, in [9] authors perform a scan of commits into some of the OSS Gilab repositories to determine the frequency of security defects in Kubernetes manifests based on the appearence of particular keywords in commit messages. The dataset size is very small and the methodology does not lead to reliable results. Thus, this article has no bearing on our paper. Paper [19] proposes a model for automatic misconfiguration detection and fix model for the containers during the deployment. The article is purely theoretical and the proposed model has no application in our case. While automatic failure mitigation is not our goal, we provide a strong base for further research on this topic as we discuss in Conclusion. Authors in [10] provide a strong research into automated anomaly detection inside the Kubernetes cluster using probabilistic state machines. Proposed method yields good results and outperforms existing ML-based solutions. Finally, probably the most relevant paper for our research is [11]. Not only

it utilizes DevOps approach, but also partly overlaps with our research. While they focus their research on Docker images, we further expand it to the Kubernetes platform and focus on the Kubernetes security tools, in particular.

Chapter 1

Introduction

In the following sections, we introduce the key concepts relevant to our research. Since topics such as containerization and container orchestration — and operations in general — are often underrepresented in the academic curriculum, we find it essential to provide the reader with foundational context.

Chapter 2 forms the core of the theoretical part of our research. It describes the methods employed, the experimental environment we created, and the rationale behind the selection of the tools used.

Chapter 3 focuses on the implementation of the security tool developed as a result of our research. This chapter takes a practical perspective and highlights the most significant engineering aspects of our solution.

Finally, Chapter 4 summarizes our findings, concludes the research, and reflects on the outcomes achieved through the development of our custom security tool.

1.1 Containerization

When talking about the Kubernetes, it is essential to be familiar with the technology of containerization. This section introduces the reader to the basics of the containerization. We start by giving a short definition, then we examine core concepts of the containerization such as container image and Docker. We also compare it to the traditional means of application deployment. Lastly, we examine the security on the container image level.

1.1.1 Overview

According to IBM, containerization is "the packaging of software code with just the operating system libraries and dependencies required to run the code to create a single lightweight executable – called a container – that runs consistently on any infrastructure" [22].

Although containers are built to be infrastructure-agnostic, there are still certain compatibility considerations to keep in mind. One significant factor is processor architecture. Containers built for a specific architecture family (e.g., arm64, amd64, or x86) are generally not cross-compatible with infrastructures built on different architectures. However, it is possible to build multi-architecture containers that support multiple processor architectures in a single image, enhancing the flexibility and portability of the containerized applications across diverse environments.

1.1.2 Container Image

Container images are software application packages, which are shipped with all required libraries, binaries and configurations. In another words, container images are lightweight and higly portable artifacts. Usually container images are stored in Container Registries, either public (like Dockerhub or Quay) or private. When an image transitions into the running state, it becomes a container.

Container images are comprised of multiple layers. At the base layer there is usually some lightweight Linux distribution. Then, each layer introduces a change in the environment, this change might concern the code or binary, runtimes, dependencies, and other objects required to run an application. Resulting image is, thus, a combination of those layers.

1.1.3 Docker

Docker is the most widely used containerization tool. It provides the tools to build, run and store containers. Docker Engine is a collective name for the Docker build toolkit. First, there is a `dockerd` or Docker Daemon, which is a server running in the backend. Secondly, the user intercts with Docker CLI or Docker Client to build and run images. Docker lient communicates with Docker Daemon through the Rest API served at the backend. Then, there is Docker Compose, which a simple orchestration tool for the containers. It can be used to compose a few containers into a system with a shared network and storage. Lastly, Dockerhub is a public container registry, where the bulk of the publicly available images are stored.

Images are built based on the Dockerfile, which is a set of instructions. Each instruction introduces a new layer to the image. Layers can then be smartly reused by the Docker Engine to build different images with similar bases. Listing 1.1 demonstates an example of the Dockerfile. Each Dockerfile starts with a `FROM` command, which specifies the base image to use for this container. The `FROM` keyword is followed by the base image name. In our case, `registry.redhat.io` is the registry address. It is followed by the repository name (`ubi8` in our case), which is separated from the registry name by a single slash and may be preceeded by the namespace. Lastly, tag

follows repository name and separated with a colon from it. Both registry address and tag are optional. In case registry name is omitted, Docker will search for the image in the Dockerhub. If the tag is not provided, Docker will fetch `latest` tag.

```
FROM registry.redhat.io/ubi8:latest
RUN dnf install nodejs && \
    useradd -u 1000 -g 1000 -M node
COPY --chown=node:node src /app
WORKDIR /app/src
USER node
RUN npm ci
EXPOSE 3000
ENTRYPOINT ["npm", "run"]
```

Listing 1.1: A simple Dockerfile for a NodeJS app.

`RUN` command is used to run commands inside the container during the build phase. All artefacts generated by the `RUN` command stay inside the contaier and can be used during runtime. In our case we are installing the necessary binaries to run our application and creating a user to run the application. `COPY` command copies the specified resource from the local machine into the container. We also are changing the ownership for the copied files. `WORKDIR` command affects the commands after it, so that they are executed from the specified directory. `USER` command changes the current user. Last `USER` command inside the Dockerfile determines under which user the main process inside the container is running in the runtime. In our case we use our newly created `node` user. The default user for the container is `root`. `EXPOSE` command ensures that a specific port on the container is open for the external communication. Lastly, `ENTRYPOINT` is one of the few ways to define the main process of the container. When main process ends, the container stops.

1.1.4 Containerization vs Virtualization

Let us discuss why containerization is the internationally accepted enterprise solution nowadays and why do software arhitects tend to choose it over traditional virtualization solutions.

Figure 1.1 provides a side-by-side comparison of a Virutal Machine and a cloud infrastructure, each with three applications deployed. We can see that each application on the container side is missing a "Guest OS" layer. Here lies the main advantage of the containers. Absence of the Guest Operating System provides a lot of advantages, which we discuss further below.

The most important advantage of the containers is their resource efficiency. Containers only include the application code and its dependencies, which makes them very

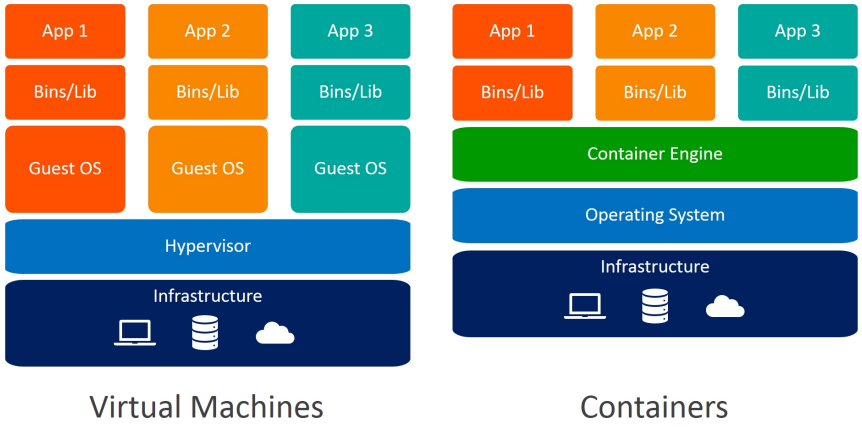


Figure 1.1: Side-by-side comparison of VM and container infrastructures [8].

small compared to the Virtual Machines, which tend to be very bulky and grow in size as development progresses. Containers share the host operating system kernel, so they consume significantly less CPU, memory, and storage than virtual machines, which require a full OS for each instance. This lightweight nature allows more containers to run on a single host, maximizing resource utilization and reducing overhead. Better resource efficiency means also lower costs for the user.

Then, startup speeds are significantly lower for the containers as they do not need to initialize the whole OS boot sequence. This feature also enables the scaling capabilities for the containers, allowing applications to respond quickly to changes in demand.

Additionally, based on [18], containers are more consistent than virtual machines. Packed with the required dependencies, they behave in the same way across different environments. As they are isolated from the OS, containers are almost immune to the compatibility issues. This gives them a strong portability advantage. They provide an abstraction that makes it easier to move workloads across various platforms.

Lastly, containers also lead when it comes to automation and CI/CD pipelines. Containers can be easily versioned, updated, and rolled back, allowing for smooth integration into CI/CD pipelines. This streamlines deployment, testing, and rollback processes, leading to faster development cycles. VMs can also be updated and rolled back, but the process is usually slower and more complex.

These are some of the most significant advantages of containerization. All of them contribute to fast build and deploy speeds, which also means high development speeds. While costing less money and providing a lot more flexibility, they become essential for successful enterprise software development. For large-scale development, test and production environment containerization has become an obvious choice over the virtualization.

1.1.5 Security Concepts

There are few security concepts native to the containerization. Security of a container starts with the secuirity of an image. When considering image metadata, image immutability is the core concept. Once a container image is built and tagged, it should not be altered. Any changes should result in a new version or tag. For further security image digests can be used. An image digest is a SHA256 hash that uniquely represents an image's content. So, for instance, a more secure way to write `FROM ubuntu:latest` would be `FROM ubuntu@sha256:abcd1234efgh5678`. Additionally, images can be signed using signing tools. Docker Content Trust is one of the implementations and can be used to ensure images are signed and verified.

Image can only be as secure as the binaries and libraries within it. It is recommended to regularly update binaries inside the image to their latest stable versions. Libraries are also a subject to regular secuirty updates. There are a lot of image scanners available that can effectively detect outdated packages and libraries inside the image. Most cloud providers nowadays incorporate such tools into their image registry services, so that all pushed images are automatically scanned for vulnerabilities.

Containers can be considered a secure environment by their nature. Being separated from the host OS, they do not share their weaknesses with the host OS. However, it is recommended to follow the rule of least privilige when dealing with the containers. They should only be granted the necessary capabilities and protected with runtime security tools like Falco, Sysdig, or AppArmor. Root containers should be always avoided.

1.2 Kubernetes

In this section we will dive into the topic of Kubernetes. We start by shortly overviewing the Kubernetes architecure. Then we expand on the Kubernetes resources, their kinds and their role in the target application environment. Finally, we consider some of the security aspects of the Kubernetes and its varieties.

1.2.1 Overview

Kubernetes, also known as K8s, is an open source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community [15].

IBM defines Kubernetes as an open source **container orchestration platform** for scheduling and automating the deployment, management and scaling of container-

ized applications. Today, Kubernetes and the broader ecosystem of container-related technologies have merged to form the building blocks of modern cloud infrastructure. This ecosystem enables organizations to deliver a highly productive hybrid multicloud computing environment to perform complex tasks surrounding infrastructure and operations. It also supports cloud-native development by enabling a build-once-and-deploy-anywhere approach to building applications [23].

It is important to emphasize that the Kubernetes abstracts the actual machines (nodes) from the user. Nodes can be physical on-premises servers, or VMs that reside either on-premises or at a cloud provider. Kubernetes takes on the responsibilty of figuring out the deployment target for a particular application. That is, user only defines the desired state of the infrastructure using YAML or JSON configuration files. Kubernetes then creates all the workloads based on the applied configuration.

1.2.2 Kubernetes Architecture

While Kubernetes requires at least three nodes to run, there are some implementation designed for the local use, which emulate this concept (see Subsection 1.2.6). The master node is called control plane. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and the cluster runs multiple nodes, providing fault-tolerance and high availability. Worker nodes host the actual workload inside the cluster. Figure 1.2 provides a simple high-level overview of a typical Kubernetes cluster with a Control Plane and three worker nodes.

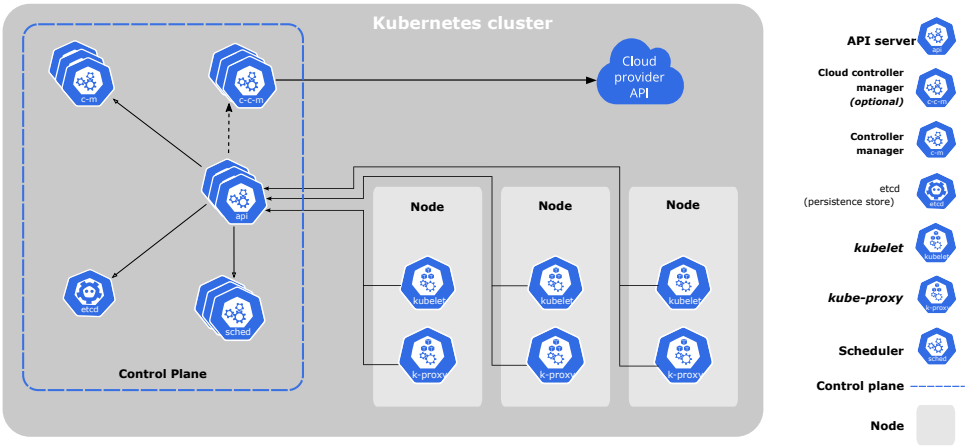


Figure 1.2: Kubernetes cluster architecure overview with its main components [14].

Control Plane overview

Control plane runs the following components:

- **kube-apiserver**
  Kube-apiserver exposes the Kubernetes API, which is acting as a frontend for the Kubernetes control plane.

- **etcd**
  Etcd is a key-value store, where all of the cluster data is stored.

- **kube-scheduler**
  Each time a new pod is created, it is passed to the kube-scheduler, which assigns the pod to the specific node to run on (based on individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines).

- **kube-controller-manager**
  Each Kubernetes resource has its own controller (e.g. NodeController, JobController, ServiceAccountController); all of them are compiled as one binary called kube-controller-manager.

- **cloud-controller-manager**
  Cloud-controller-manager embeds cloud-specific control logic. It differs depending on the cloud provider or can be absent completely, when running Kubernetes locally.

Worker Node overview

Each worker node has a kubelet and kube-proxy installed. Kubelet is an agent that manages runnning pods and containers. Kube-proxy is a network proxy that implements parts of the Kubernetes service concept. It maintains network rules on nodes, making in- outside-cluster communitcation possible.

Then, of course, each worker node has a set of running pods. A typical use would involve multiple running applications. Depending on the size of the node and the application resoucre consumption it can accomodate on average from one to a few dozens applications with various business purposes.

1.2.3 Kubernetes Resources

Kubernetes resources are fundamental components that define various entities within a Kubernetes cluster. Resources are objects that represent the desired state and configuration of the infrastructure, applications, and services running on the cluster. Kubernetes provides a range of resources that enable developers and operators to define, manage, and scale containerized applications, network policies, storage requirements,

and more. These resources are defined declaratively in YAML or JSON files, which makes infrastructure setup consistent and reproducible.

Among key Kubernetes resoucres are:

- **Pods**
  Pod is the atomic workload unit in the Kubernetes cluster. It encapsulates one or more containers that share the same network. It represents a single instance of a running application.

- **Deployments**
  Deployments are a higher level of abstraction for the Pods. They allow to define replica count and rollout/rollback strategy for the updates, which can be used to ensure availability for the application.

- **Services**
  Services provide a communication layer for the pods inside one cluster. Being an abstraction over the pods' network, they provide reliable access to the selected workloads, while serving as a Load Balancer.

- **ConfigMaps** and **Secrets**
  ConfigMaps and Secrets allow users to store data outside the workload. ConfigMaps are usually used to store non-sensitive information like environment and application configuration parameters. Secrets are a more secure resource designed for API keys, passwords and other sensitive data.

- **PersistentVolumes** and **PersistentVolumeClaims**
  These resources enable stateful applications to request and mount durable storage within a cluster, allowing data to persist independently of the Pod lifecycle.

Above are the most commonly used resources, which we also leverage in the practical part of the paper. Therefore, it is important that the reader understands the position and the purpose of each resource in the cluster infrastructure.

Workloads

Minimal computing units in Kubernetes are Containers, which are running in Pods. However, to simplify the management of Pods, Kubernetes has workload resources, which manage the set of Pods. They make sure the desired number of Pods of right kind are running to match the declaration.

Deployments and ReplicaSets are a good fit for stateless applications. Each pod in the Deployment is interchangebeable. Deployments are easily scalable and have built-in versioning and rollout mechanisms.

StatefulSet allows to create sets of stateful applications. They might share the same PersistentVolume and replicate data between each other.

DaemonSet defines Pods that provide node-local facilities. These might be fundamental to the operation of your cluster, such as a networking helper tool, or be part of an add-on.

Job and CronJob define tasks that run to completion and then stop. Jobs represent one-off tasks, whereas CronJobs recur according to a schedule.

Networking

Kubernetes networking model makes Pods look like VMs in the networking aspect. Pods on any nodes can communicate with each other without NAT. Containers inside the same Pod share its network meaning that they can reach each other using localhost.

Kubernetes networking addresses four concerns:

- Containers within a Pod use networking to communicate via loopback.
- Cluster networking provides communication between different Pods.
- The Service API lets you expose an application running in Pods to be reachable from outside your cluster.
- Ingress provides extra functionality specifically for exposing HTTP applications, websites and APIs.
- You can also use Services to publish services only for consumption inside your cluster.

Storage

Kubernetes does not ship a particular implementation of storage. However, it provides a range of resources that define the storage concept and supports different types of volumes. A Pod can use any number of volume types simultaneously. Ephemeral volume types have a lifetime of a pod, but persistent volumes exist beyond the lifetime of a pod. When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes. For any kind of volume in a given pod, data is preserved across container restarts.

At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

PersistentVolumes and PersistentVolumeClaims are the resources kinds most important to understand here.

- A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It

is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

- A PersistentVolumeClaim (PVC) is a request for storage by a user. In some ways, is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany). Once and Many here refer to a number of Pods that can perform the read or write simultaneously.

1.2.4 Role Based Access Control

Role-Based Access Control, **RBAC** in short, is a fundamental security mechanism in Kubernetes that governs access to resources within a cluster. It enables administrators to define policies that restrict or allow actions based on the roles assigned to users, groups, or service accounts. Proper configuration of RBAC is critical to ensuring a secure Kubernetes environment.

There are a few components of RBAC that allow us to create users and define their access permissions and restrictions.

- **Roles**
  A Role defines a set of permissions within a specific namespace. It is used to control access to resources like Pods, ConfigMaps, or Deployments in that namespace.

- **ClusterRoles**
  A ClusterRole, on the other hand, is a cluster-wide version of a Role. It applies to resources that span the entire cluster, such as nodes or PersistentVolumes, or to resources in all namespaces.

- **RoleBindings**
  A RoleBinding associates a Role with one or more subjects (users, groups, or service accounts) within a specific namespace.

- **ClusterRoleBinding**
  A ClusterRoleBinding extends this association to ClusterRoles and applies it cluster-wide.

RBAC policies are defined using YAML manifests. The primary fields in an RBAC policy include:

- **APIGroups**: Specifies the API group of the resources.
- **Resources**: Indicates the type of resources, such as Pods, Services, or Deployments.
- **Verbs**: Lists the actions allowed, such as get, list, create, or delete.

The symbol '*' is a wildcard that might be used to represent "all" for specific fields. However, its use should generally be avoided in compliance with least privilege principle. Listing 1.2 defines a `pod-reader` **Role** inside the `my-namespace` namespace.

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

Listing 1.2: An example of a Kubernetes Role definition.

1.2.5 Data Security

Kubernetes provides only one kind of resource to manage sensitive data. A Kubernetes Secret is designed to store sensitive data in a way that prevents it from being exposed unnecessarily. API tokens, database credentials or encryption keys can be stored as Secrets. By default, Secrets are stored unencrypted in etcd. Enabling encryption at rest is crucial to secure their contents. For a more secure solution external tools such as HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault should be considered as an alternative to the Kubernetes Secrets to securely manage and rotate sensitive information.

On the topic of the filesystem security of the pods, Kubernetes itself does not handle encryption at the storage layer but relies on the underlying storage system to provide it. Appropiate namespace isolation coupled with a well-defined RBAC rules make Secrets and PVCs sufficiently secure for a common use case.

1.2.6 Other Kubernetes Implementations

Openshift

Openshift is another container orchestration platform built on top of Kubernetes by Red Hat. Openshift extends the basic Kubernetes functionality and emphasizes on

security and automation. It ships with a lot of additional enterprise-level features like image registry, pipeline operators and OpenShift Web Console.

OpenShift emphasizes security with stricter security defaults, such as non-root containers by default and integrated role-based access control. It also provides Security Context Constraints for fine-grained control over workloads. They control the actions that a pod can perform and what it has the ability to access. Imagine we have a ServiceAccount `mysvcacct` inside `default` namespace (or project, as they called in Openshift). Then if we would like to grant a pod inside the `default` namespace running under `mysvcacct` ServiceAccount, we could do so using this command: `oc adm policy add-scc-to-user privileged system:serviceaccount:myproject:mysvcacct` .

The OpenShift dashboard, part of the OpenShift Web Console, provides a graphical user interface for managing and monitoring Kubernetes resources within an OpenShift cluster. It simplifies cluster management and enables users to perform tasks without extensive use of the command line. Overview page of the dashboard can be seen on Fig 1.3.

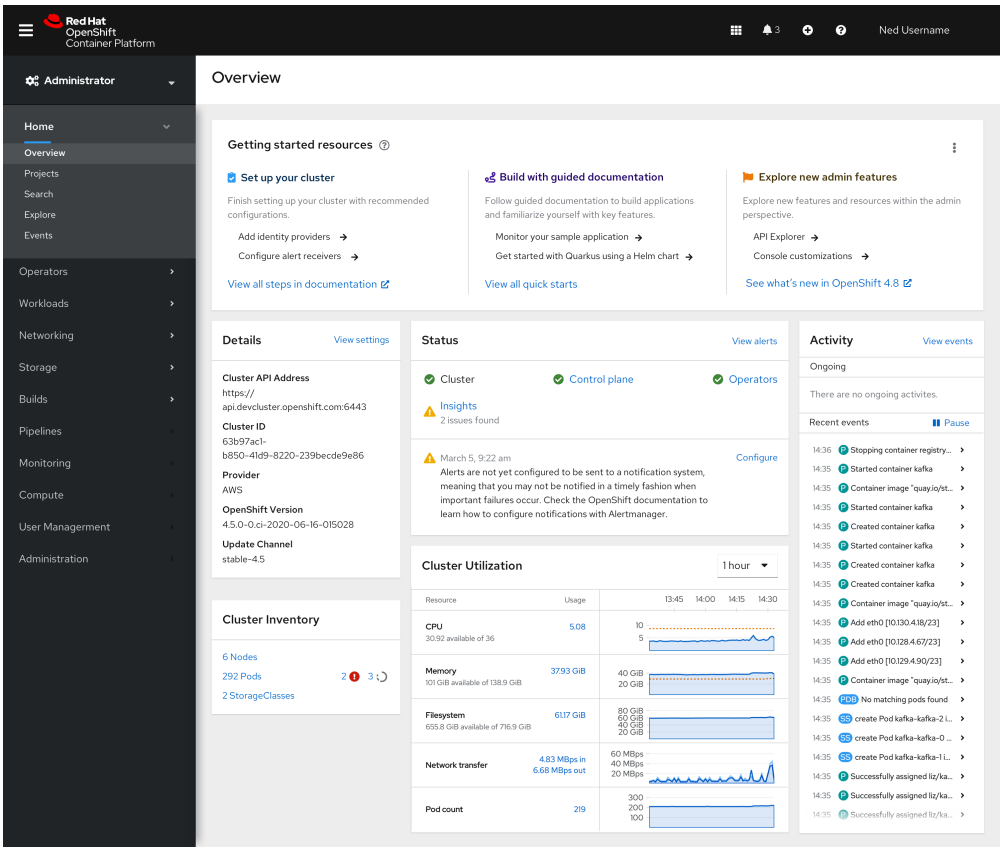


Figure 1.3: Openshift dashboard. Overview page.

Rancher Desktop

Rancher Desktop is "an open-source application that provides all the essentials to work with containers and Kubernetes on the desktop" [17].

Basically, it provides a local Kubernetes cluster with convenient GUI and a dashboard by default. It is widely used by developers to test their application in cloud environments without commiting the changes.

Big advantage of Rancher Desktop is its simple installation process. Rancher Desktop is available for Windows, Linux and MacOS and is also bundled with all the necessary CLI tools like docker, nerdctl, kubectl, helm, and others.

Rancher Desktop allows to choose from dockerd or containerd as a container backend. Additionally, users can switch between Kubernetes versions easily to ensure compatibility with various environments. GUI allows to inspect containers and images, troubleshoot the cluster and forward container ports to the local machine. Figure 1.4 captures a "Port Forwarding" page of Rancher Desktop app.

Being free and easy to setup, Rancher Desktop became our main tool for local experiments. Most of our research was done on a Rancher local cluster.

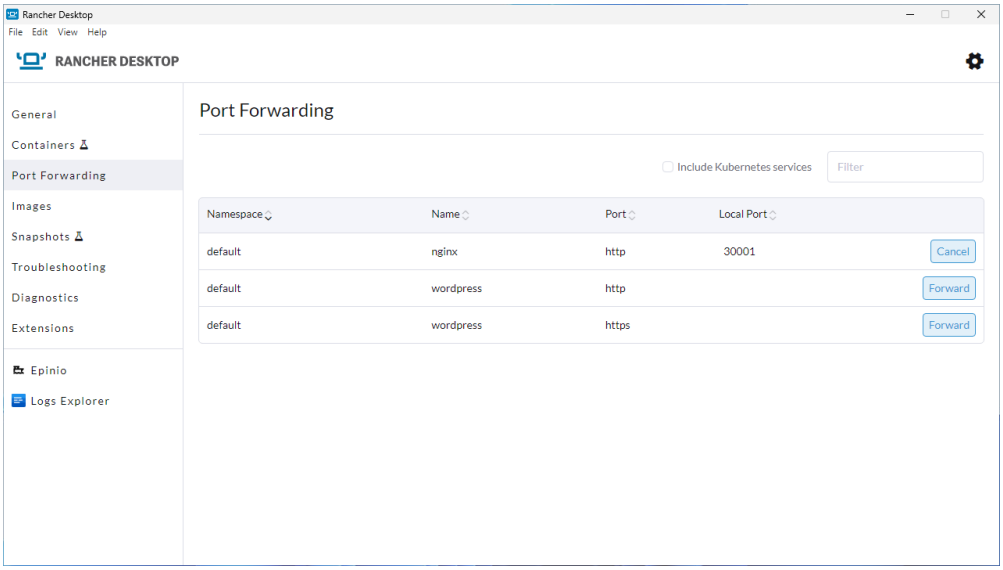


Figure 1.4: Rancher Desktop GUI.

1.3 Security frameworks

1.3.1 Overview

When we are talking about the infrastructure security of Kuberntes environment inside the cloud, we must consider multiple layers. Going from the top to the bottom, we

start with the security measurements taken on the cloud provider side. In most cases this is not something we can affect and the security of different cloud providers varies significantly. However, all of the "big five" cloud providers (AWS, Azure, Google Cloud, Alibaba and IBM) maintain high security standarts and security risks generally should not be a concern for their end users. Then, we get to the cluster itself. On the cluster level we must consider the security of the nodes, security of the cluster components and their configuration. At this layer we have already some space for the misconfigurations to appear. Here we can evaluate the security of the single components using some of the security scanners presented in Kubernetes security automation. But it should be noted that the cluster can only be as secured as its nodes. Therefore, attention should be given to the node security first. Host operating system should be regularly patched with the most recent security patches. Lastly, we get to the application level, where the security of the application code, Kubernetes resoucre configuration, libraries, dependencies and base images is the main concern. Again, this is the layer, where the developers have the most access to, thus, providing a lot of space for the possibility of a human error. In this paper we mostly work on this level when we do our research.

Over the years a variety of Kubernetes security frameworks have been developed. There are a few that target specific layers of the Kubernetes environment, but most of them are designed to assess the security on a combination of those layers. In the Kubernetes scope, security framework is a structured set of guidelines and policies designed to assess, implement and maintain security controls across different Kuberntes layers. Kubernetes security frameworks are modular (as they are usually split across different domains) and provide guidelines to minimize risks or remediation instructions to resolve the threat. Among the most well-known Kubernetes security frameworks are CIS Kubernetes Benchmark, NSA Framework and MITRE ATT&CK Framework. Among other notable frameworks are NIST 800-53 and SOC 2. The former is a comprehensive security and privacy control framework developed by the National Institute of Standards and Technology. The latter is a compliance framework designed by the AICPA, which focuses on how companies manage customer data. Payment card data security is considered by the PCI-DSS framework and the clusters used for processing card data must comply. GDPR also applies to the Kubernetes deployments if they process personal data of EU citizens. HIPAA is a framework that regulates the privacy and security of health information. Clusters hosting applications that handle ePHI must enforce encryption, logging, access control, and backup policies. ISO/IEC 27001 is an international standard for establishing, implementing, maintaining, and continuously improving an Information Security Management System, which can be applied to the Kubernetes as well.

Officially, The Kubernetes project itself (and CNCF) publish some security guidelines for each of the layers. However, a more thourough and complete list of Kuber-

netes security checks are provided by the Center of Internet Security, National Security Agency and MITRE Corporation. Below we present an overview of the Kubernetes security recommendations by CNCF, and an overview of the most popular security frameworks.

1.3.2 Kubernetes security recommendations

This section gathers the official security recommendations provided by the Kubernetes. They provide a list of concerns for each level of the cloud infrastructure. Cloud infrastructure can be viewed as a composition of four layers as displayed by the Fig 1.5.

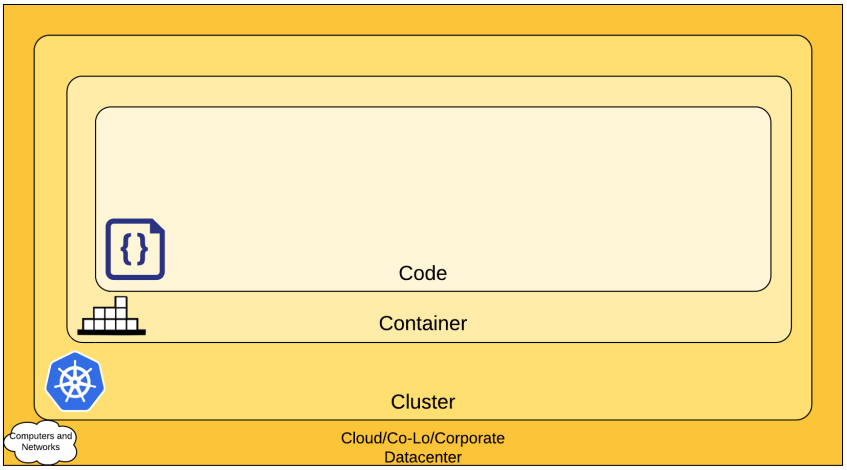


Figure 1.5: Four layers of the cloud infrastructure.

Each layer is built upon the previous one and its security depends on the security of the outer layers. It is, therefore, important to maintain high security standarts on base levels (cloud, cluster, container).

1. **Cloud**
   Each cloud provider has its own security policies and guidelines. There are, however, some general infrastructure-level security best advice described by in the Tab 1.1.

Area of Concern for Kubernetes Infrastructure	**Recommendation**
Network access to API Server (Control plane)	All access to the Kubernetes control plane is not allowed publicly on the internet and is controlled by network access control lists restricted to the set of IP addresses needed to administer the cluster.
Network access to Nodes (nodes)	Nodes should be configured to only accept connections (via network access control lists) from the control plane on the specified ports, and accept connections for services in Kubernetes of type NodePort and LoadBalancer. If possible, these nodes should not be exposed on the public internet entirely.
Kubernetes access to cloud provider API	Each cloud provider needs to grant a different set of permissions to the Kubernetes control plane and nodes. It is best to provide the cluster with cloud provider access that follows the principle of least privilege for the resources it needs to administer.
Access to etcd	Access to etcd (the datastore of Kubernetes) should be limited to the control plane only. Depending on your configuration, you should attempt to use etcd over TLS.
etcd Encryption	Wherever possible it's a good practice to encrypt all storage at rest, and since etcd holds the state of the entire cluster (including Secrets) its disk should especially be encrypted at rest.

Table 1.1: Security recommendations for the cloud layer.

2. **Cluster**

   There are two cluster security concerns that could be addressed: securing the configurable cluster components and securing the applications running in the cluster.

   There are a few things to consider regarding the application security:

   - RBAC Authorization (Access to the Kubernetes API)
   - Authentication
   - Application secrets management (and encrypting them in etcd at rest)
   - Ensuring that pods meet defined Pod Security Standards

- Quality of Service (and Cluster resource management)
- Network Policies
- TLS for Kubernetes Ingress

3. **Container**
   Securing containers is a vast topic, which deserves its own chapter. There are, nevertheless, a few general recommendation provided by the Kubernetes, which you can find in the Tab 1.2.

Area of Concern for Containers	**Recommendation**
Container Vulnerability Scanning and OS Dependency Security	As part of an image build step, you should scan your containers for known vulnerabilities.
Image Signing and Enforcement	Sign container images to maintain a system of trust for the content of your containers.
Disallow privileged users	When constructing containers, create users inside of the containers that have the least level of operating system privilege necessary in order to carry out the goal of the container.

Table 1.2: Security recommendations for the Container layer.

4. **Code**
   When it comes to code, the developers have the most flexibility to design secure applications. There are a lot of issues to address, which may vary significantly from application to application depending on its purpose, architecture and framework base. Kubernetes documentation gives a handful of recommendations regarding this topic, which are displayed below in the Tab 1.3.

Area of Concern for Code	**Recommendation**
Access over TLS only	If your code needs to communicate by TCP, perform a TLS handshake with the client ahead of time. With the exception of a few cases, encrypt everything in transit. Going one step further, it's a good idea to encrypt network traffic between services. This can be done through a process known as mutual TLS authentication or mTLS which performs a two sided verification of communication between two certificate holding services.
Limiting port ranges of communication	This recommendation may be a bit self-explanatory, but wherever possible you should only expose the ports on your service that are absolutely essential for communication or metric gathering.
3rd Party Dependency Security	It is a good practice to regularly scan your application's third party libraries for known security vulnerabilities. Each programming language has a tool for performing this check automatically.
Static Code Analysis	Most languages provide a way for a snippet of code to be analyzed for any potentially unsafe coding practices. Whenever possible you should perform checks using automated tooling that can scan codebases for common security errors.
Dynamic probing attacks	There are a few automated tools that you can run against your service to try some of the well known service attacks. These include SQL injection, CSRF, and XSS. One of the most popular dynamic analysis tools is the OWASP Zed Attack proxy tool.

Table 1.3: Security recommendations for the Code layer.

1.3.3 CIS Kubernetes Benchmark

CIS publishes a variety of documents for an array of different platforms and infrastructure components. Benchmarks are available for 10 cloud providers, 26 operating systems, 19 kinds of system software, as well as for mobile devices, network devices and desktop software. Benchmarks for the cloud services, for instance, include such cloud providers as Alibaba, Amazon, Google, IBM, Microsoft, Oracle and a few others. For Kubernetes platform, specifically, GKE, AKS, EKS and general Kubernetes

benchmarks are available for non-commercial use.

CIS Benchmark is a PDF file, which contains a list of checks (both manual and automated), that can be performed to ensure the most secure environment. CIS Benchmark file for Kubernetes has 5 categories of security recommendations:

1. Control Plane Components
2. etcd
3. Control Plane Configuration
4. Worker Nodes
5. Policies

There are multiple subcategories for the most of the categories as well. Each recommendation includes the general information about the check, such as title, description and assessment status. Audit procedure describes the steps necessary to determine if the target system is compliant with the recommendation. Additionaly, every recommendation is supplied with the rationale statement, which gives a "Detailed reasoning for the recommendation to provide the user a clear and concise understanding on the importance of the recommendation." Finally, remediation procedure provides instructions for bringing the the target system into compliance with the recommendation.

CIS Benchmark document [13] states that "'All CIS Benchmarks (Benchmarks) focus on technical configuration settings used to maintain and/or increase the security of the addressed technology, and they should be used in conjunction with other essential cyber hygiene tasks ... In the end, the Benchmarks are designed to be a key component of a comprehensive cybersecurity program."' This means that even though benchmarks cover many aspects of Kubernetes security, authors recommend to use in combination with other defensinve and monitoring tools.

1.3.4 NSA Framework

The NSA Kubernetes Hardening Guide is a security framework jointly published by the U.S. National Security Agency and CISA. It provides actionable recommendations to secure Kubernetes clusters against real-world threats. As stated in the document itself [7], it "describes the security challenges associated with setting up and securing a Kubernetes cluster. It includes strategies for system administrators and developers of National Security Systems, helping them avoid common misconfigurations and implement recommended hardening measures and mitigations when deploying Kubernetes". First released in August 2021, it was last updated in August 2022, but is still used worldwide as the most of the described concepts are still applicable to the newest Kubernetes versions.

This guide is a well-structured document 66 pages long. It is written as a text guide and divided into multiple sections. The sections are Kubernetes Pod Security,

Network Separation and Hardening, Authentication and Authorization, Audit Logging and Threat Detection, Upgrading and Application Security Practices. Each section describes the best security practices for a particular domain. For instance, the document proposes a hardened container build workflow, where each image must pass three security services (image signature verification, image scanner, configuration validation) before being deployed into the cluster. Figure 1.6 displays a schematic overview of the workflow. Interstingly, the document references and is partially based on MITRE ATT&CK Framework.

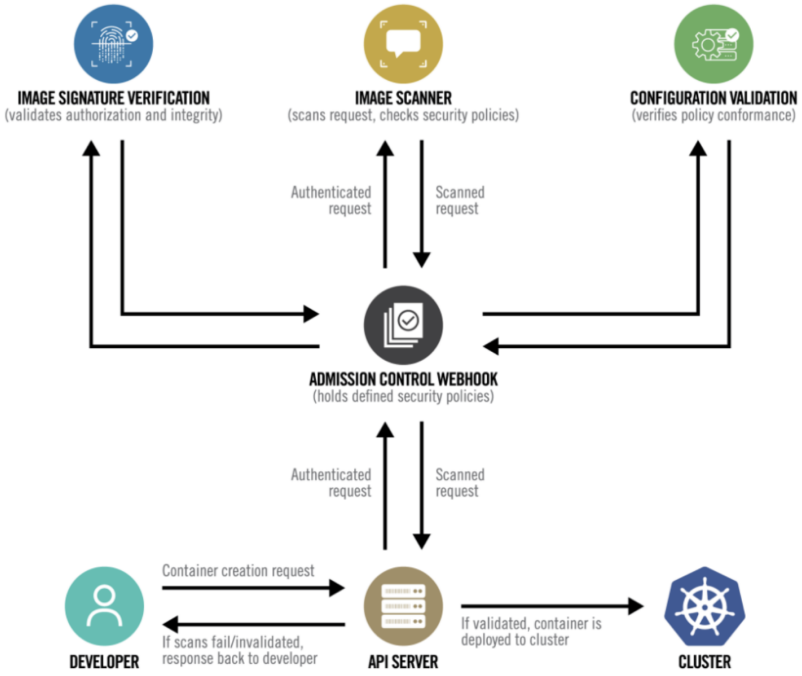


Figure 1.6: A hardened container build worflow.

1.3.5 MITRE ATT&CK Framework

The MITRE ATT&CK Framework (Adversarial Tactics, Techniques, and Common Knowledge) is a comprehensive knowledge base of cyber adversary behavior. It categorizes the tactics and techniques that attackers use across various stages of an intrusion, helping the security specialists to understand, detect, and mitigate real-world threats. Tactics describe the rationale and goals of the attacker, among the most common tactics are Initial Access, Execution, Discovery and Impact. Techniques describe the methods used by the attackers to achieve a tactic. It can be any action from Phishing to Exploitation for Privilege Escalation. Techniques are further granulated into

sub-techniques. Finally, Mitigations provide recommendations and controls for each technique while Detection describes the patterns, which can be used to identify technique usage via logs, events and telemetry.

Inside the Kubernetes context, MITRE ATT&CK Framework publishes techniques for the containerized environments domain. They focus on Kubernetes control plane, worker nodes, pods, and related infrastructure and map known attack behaviors to Kubernetes-specific contexts. For instance, the Initial Access tactic is mapped to the such subtechniques as Compromised Kubeconfig, Exposed Dashboard/API, Impact tactic makes use of Data Destruction (like deleting pods or nodes), Denial of Service and Resource Hijacking techniques. Figure 1.7 displays the MITRE ATT&CK matrix for the Container platform.

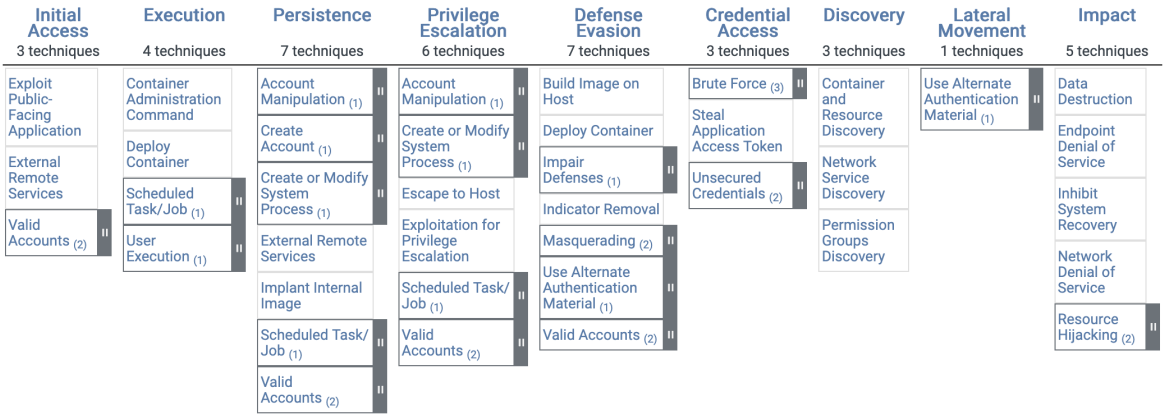


Figure 1.7: MITRE ATT&CK matrix for the Container platform [12].

Chapter 2

Research

2.1 Kubernetes security automation

This section introduces the reader to the topic of the security automation inside the Kubernetes cluster. We discuss different security tools, their place in the cloud Infrastructure and examine their usage patterns. Additionally, we explain why were the specific tools chosen for our research.

2.1.1 Overview

Kubernetes security scanners and operators provide an array of defensive capabilties. Most of them act in the form of informator. That is, they do not perform any remediatory actions, but only provide user with information about the cluster security status. Nevertheless, there are some solutions on the market that are capable of resolving some of the security issues automatically. This, on the other hand, introduces another layer of concern: can we really trust a third-party system to introduce modifications to our infrastructure? That the former type is the most abundant and is the focus of this paper. Automatic remediation can be then implemented as a part of CI/CD pipeline based on the scan results. This ensures that it is compliant with the company's policies and is tailored to the company's needs.

One of the ways to classify Kubernetes security scanners is by the scan target. Here we can roughly divide them into three groups: configuration file scanners, cluster scanners and container image scanners. Most of the tools, however, can be put into multiple different groups. Cluster scanners usually are able to perform container image scanning as well and it is a part of the full cluster scan. Cluster scanners detect misconfigurations in the cluster infrastructure and its essential components. They look for, among other things, containers running with extensive privilages, exposed sensitive workloads and plaintext secrets. Container image scanners look for known vulnerabilities inside the images. Configuration file scanners perform a scan of the cluster configuration files.

For large infrastructure with a number of applications deployed there might be over several tens of thousands lines of configuration and such scanners aim to detect any known misconfigurations by going through theese lines.

Cluster security scanners can be further classified by the execution point. Again, there is usually more than one way to run a scan, but here are a few options: run a scanner tool as a container, run it from a remote machine connected to the cluster, run it is an operator, which can perform scan automatically on a regular basis. To always keep cluster up-to-date with the most recent security patches the best solution would be to either install an operator or integrate a security scanner tool into your CI/CD pipeline.

2.1.2 Selection Criteria

To perform our assessment we have chosen from a variety of Kubernetes security scanners. Though the area is still relatively new, there is a variety of tools with different purposes available on the market. We made our choice based on the following criteria:

- **free-to-use**
  We do include some proprietary tool testing further in our research as an additional comparison, however, for the main part we only use free tools. Kubernetes itself is distributed under Apache License 2.0, which means it is inherently free to use. The ability to adopt these tools without financial constraints enables wider adoption, thus, contributing to the community-driven innovations. Finally, this research not being funded, we could only afford to work with openly distributed software.

- **open-source**
  Again, we are sticking to the open-source nature of the Kubernetes. By selecting open-source tools, this research ensures that each tool's codebase is transparent and can be reviewed by security experts. This transparency increases trust in the tools' effectiveness, as the community can spot, disclose, and even patch any vulnerabilities in the software. Another advantage is the customization of the open-source software as the companies can adapt the tools to their specific Kubernetes security needs.

- **designed with cloud in mind**
  Designed to be used in the Kubernetes environment specifically, these tools should offer features like scanning container images for vulnerabilities, but also monitoring network policies, securing Kubernetes configuration files, or identifying misconfigurations within clusters. Tools built specifically for Kubernetes are more

efficient, as they are optimized to address the distinct aspects of the platform, making security management more effective.

- **has an active community support**
  Tools with active communities tend to have more frequent updates, faster response times for bug fixes, and a wide range of contributors who bring diverse insights to improve functionality and security. The world of the software security is changing rapidly and an active community means that the tool is up-to-date with the most recent events. A thriving community also means that users can easily access support on the community forums.

Based on the aforementioned criteria we ended up choosing and testing the following tools:

- Trivy
- Kube-bench
- Prowler
- Kubescape

In the next chapters we closely examine each selected tool and explain how it is matches our selection criteria. Additionally, we compare them to each other and highlight their strong and weak sides.

2.1.3 Trivy

Trivy is an Aqua Security open source project with a vast array of use cases. It supports multiple scan targets and includes multiple scanners. Among the supported targets are:

- Container Image
- Filesystem
- Git Repository
- Virtual Machine Image
- Kubernetes

Trivy includes scanners for:

- OS packages and software dependencies in use (SBOM)
- Known vulnerabilities (CVEs)
- IaC issues and misconfigurations
- Sensitive information and secrets
- Software licenses

According to the Trivy official Gihub page [5], it can be installed on the local machine using any of the popular package mangager or by downloading a binary from the Github Releases. It can also be ran as a Docker container or Kubernetes Operator. Furthermore, Trivy can be integrated into GitHub Actions or installed as a Visual Studio Code plugin. Aqua Security uploads each new release as a Docker image into the Dockerhub repository. There is a variety of supported configuration parameters for Trivy Kubernetes scanning feature. Users can specify which scanners to include, which namespaces to skip, which nodes to scan and the format of the output. An example of a Trivy misconfiguration scan command executed against the default Kubernetes context, which would output a short summary of findings and skip **dev-system** namespace, is included below (see Listing 2.1).

```
$ trivy k8s \
    --scanners=misconfig \
    --report=summary \
    --exclude-namespace=dev-system
```

Listing 2.1: An example of a Trivy Kubernetes scan command.

Since Kubernetes is listed as a natively supported target and Trivy can scan for both vulnerabilities and misconfigurations, Trivy is well-suited for our research. It is also open source and available for free. Presently, Trivy's Github repository has over 2800 issues, with a little over than 150 of them being open, repository's commit history shows active development with a bi-monthly minor release cycle and yearly major release cycle. Thus, we can assume an active community and developer support.

2.1.4 Kube-bench

Kube-bench is another open source tool developed by Aqua Security. Their Github page [6] states that it checks whether Kubernetes is deployed securely by running the checks against the CIS Kubernetes Benchmark (see 1.3.3). Kube-bench is designed specifically for Kubernetes. Users can run the it inside a Docker container or deploy it as a Kubernetes job, however, there is still an option to download the binary on the local machine and run it against the desired Kubernetes cluster.

Since kube-bench has a much narrower feature set than Trivy, it is much more simple in usage, but still highly configurable. Configuration can be supplied via a config file or we can pass the configuration parameters directly using one of the 24 flags. An example of a simple scan command targeting **master**, **node**, **etcd**, **policies** CIS Benchmark categories is provided in Listing 2.2.

```
$ kube-bench run \
    --targets master,node,etcd,policies
```

Listing 2.2: An example of a Kube-bench scan command.

Kube-bench is actively supported by the developers and the community. At the present moment, Kube-bench Github repository has about 500 issues, 10% of which are currently open. Repository receives updates on a weekly basis and the new version is released monthly.

2.1.5 Prowler

Prowler, as described on the Github page [3], is an open source security tool designed to perform Kubernetes security best practices assessments, audits, incident response, continuous monitoring, hardening and forensics readiness, and remediation. It is shipped with a built-in dashboard, which displays scan results in graphical format (see Fig 2.1). However, the dashboard can only read the scan results from the folder on the host machine and the user cannot trigger a new scan directly from the dashboard. Users have to install a separate Prowler App inside the clusters to trigger the scan. According to the documentation [24], "it provides a user-friendly interface to configure and run scans, view results, and manage your security findings."

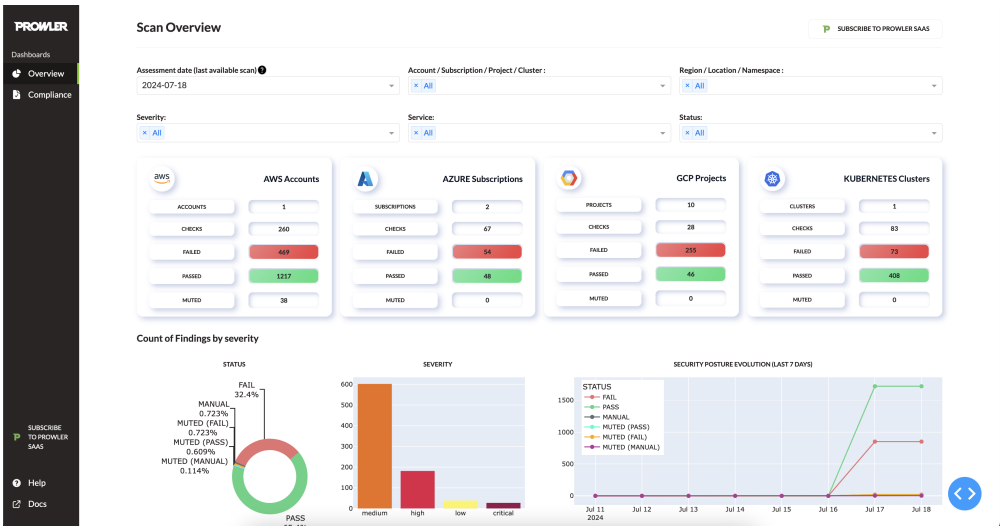


Figure 2.1: Prowler dashboard interface.

Prowler CLI supports Azure, Google Cloud, AWS and generic Kubernetes providers. It performs a scan against multiple Kubernetes security frameworks to ensure the best coverage. Prowler supports output in CSV or JSON-OCSF format, the latter being an independent open source developed JSON schema by Open Cybersecurity

Schema Framework project, which has recently joined the Linux Foundation organization. Again, we configure the Prowler job either via a configuration file or by passing the flags directly in the execution command. Listing 2.3 provides an example on how can a Prowler scan be triggered. This particular execution would output the results in **json-ocsf** format, including only failed and manual checks.

```
$ prowler kubernetes \
    --status FAIL,MANUAL \
    --output-formats json-ocsf
```

Listing 2.3: An example of a Prowler scan command.

There are over a thousand issues in official Prowler Github repository. The development is still in progress and the tool receives regular updates. The project has an active community with operational development support.

2.1.6 Kubescape

Kubescape is an open-source security platform designed to harden the security of a Kubernetes cluster. Kubescape CLI allows users to scan the cluster from the local machine. Kubescape Operator enables image vulnerability scanning as well as continuous and scheduled scanning. Kubescape also offers Github Actions integration for CI/CD pipelines according to the official Github page [2].

Kubernetes cluster is verified against a posture library (available at Github [4]), which is a collection of frameworks each containing a set of controls. By default, the library is comprised of NSA Framework controls, MITRE ATT&CK Framework controls and CIS Benchmark controls.

Scan configuration is again passed with multiple flags. There are flags allowing to select the namespaces to include into scan, namespaces to exclude from the scan, output format and location. Scanning can be then performed using the command displayed in Listing 2.4.

```
$ kubescape scan \
    --format json \
    --output kubescape-scan.json
```

Listing 2.4: An example of a Kube-bench scan command.

Unfortunately, when executed in this mode, Kubescape performs only a few checks. In order to perform complete vulnerability and misconfiguration scan, Kubescape requires Kubescape Operator installed in the cluster.

Kubescape's repository shows ongoing development with regular update releases.

2.1.7 Preliminary Comparison

This section compares the selected tools based on their declared feature set, ease of installation and execution, scan targets, supported cloud providers and security frameworks. This comparison does not include the analysis of the scan results, performance or any other aspects of execution. A detailed analysis of the generated reports can be found in Chapter 4.

We start by comparing the scanners by their feature set. Trivy has the most broad feature set as it is able to scan for vulnerabilities inside the images, perform IaC scan (supporting Terraform, Helm, plain Kubernetes YAML), scan for misconfigurations in Kubernetes resources and generate SBOM. Kube-bench is designed to perform only two tasks: CIS Kubernetes Benchmark checks and Node-level audit, making it very specialized tool. Prowler is able to detect misconfigurations and security threats, perform CIS Benchmark checks and cloud security audit. Kubescape's features include misconfiguration and container vulnerability scanning, RBAC risk analysis and SBOM generation. While all of the scanners can detect Kubernetes misconfigurations, only Trivy and Kubescape have the ability to scan for the vulnerabilities in containers. SBOM generation is a requirement for enterprise-level scanners nowadays, which makes Kubescape and Trivy stand out once again.

When we compare the tools by the supported cloud providers, simplicity of Kube-bench makes it the leader in this category. Since Kube-bench only performs static analysis, this makes it cloud agnostic, meaning that it can be used with any cloud provider as long as the provider offers Kubernetes services. Trivy, Kubescape and Prowler all support the same cloud providers. Kubescape and Trivy both natively support AWS, Azure and GCP. Prowler's focus is AWS, but it also supports Azure and Google Cloud Platform.

All of the presented tools declare full CIS Benchmark coverage, which is the only supported framework for Kube-bench. Trivy additionally declares NSA and MITRE ATT&CK coverage. Kubescape further extends the set with NIST 800-53 and SOC 2 frameworks, which are more specialized frameworks, the former developed by the U.S. government and focuses on technical and administrative controls, while the latter focuses on the customer data security and compliance. Prowler mixes the CIS Benchmark checks with the four major security and privacy frameworks or regulations relevant to organizations handling sensitive data (PCI-DSS, GDPR, HIPAA, ISO27001).

From the user perspective we can compare the scanners by the ease of installation and execution. Trivy's binary is available for download using most of the popular package managers (like Brew for MacOS and apt-get, yum, pacman and others for various Linux distros). Less secure but more convenient way of installation is by using a script. Additionally, Trivy has a Docker image available in Docker Hub, GitHub

Container Registry and AWS Elastic Container Registry. Trivy can be executed against an active Kubernetes context using the binary or installed as an operator inside the cluster for automatic scanning every six hours. Kube-bench can only be run from inside the container. Kube-bench provides a **job.yaml** file, which can be used to run it inside the cluster as a Kubernetes Job. Kube-bench runs checks specified in controls files that are a YAML representation of the CIS Kubernetes Benchmark checks. Kubescape supports the usual installation channels. Additionally, users are able to download scan artifacts (frameworks) separately. Kubescape Operator can be installed inside the using a Helm chart. Unfortunately, Kubescape Operator must be installed in the cluster for the Kubescape CLI to be able to scan for vulnerabilities. Prowler CLI can be installed only as a Python module, but there are also Docker images available to download from the Docker Hub and AWS Public ECR. Prowler also provides an applciation, which displays scan results in a graphical format. Table 2.1 summarizes our preliminary comparison findings.

Tool	**Features**	**Cloud support**	**Frameworks**	**User experience**
Trivy	Vulnerability scan, Misconfiguration scan, IaC scan, SBOM generation, Operator	AWS, Azure, GCP	CIS, NSA, MITRE ATT&CK	CLI tool
Kube-bench	CIS Benchmark audit	cloud agnostic	CIS	Kubernetes Job
Prowler	Cloud audit, misconfiguration scan, compliance scan, Operator	AWS, Azure, GCP	CIS, PCI-DSS, GDPR, HIPAA	CLI tool, UI app
Kubescape	Operator, misconfiguration scan, vulnerability scan, SBOM generation, RBAC analysis	AWS, Azure, GCP	CIS, NSA, MITRE ATT&CK, NIST 800-53, SOC 2	CLI tool

Table 2.1: Kubernetes security scanners preliminary comparison.

2.2 Methodology

Research aims to assess scanner capabilties by performing quantative and qualitative analysis of the scan results against misconfigurations and vulnerabilities inside the cluster. In order to do so, a cluster environment populated with a variety of misconfigurations and vulnerabilities was prepared. Refer to the Section 2.4 for the list of implemented vulnerabilities. Since the number of misconfigurations is known beforehand, no baseline is needed. Then, each selected security tool is executed against the cluster and the results are parsed. For the qualitative analysis we count the amount of vulnerabilities found, number of the missed categories, the amount of false positives and the number of vulnerabilities in each severity group. For the qualitative analysis we determine if the most critical misconfigurations were detected and assess how well the tool reports relevant details. This includes evaluating whether the scan output provides sufficient context for remediation, such as affected resources, potential impact, and suggested fixes.

To ensure fairness in comparison, all scanners were executed under the same conditions. Each tool was configured according to its documentation. Any limitations in scanner capabilities, such as an inability to scan certain Kubernetes objects or container runtime restrictions, are documented in further chapters.

After obtaining scan results, a comparative analysis was conducted to identify patterns in detection capabilities across different tools.

The findings from both quantitative and qualitative evaluations form the basis for determining if existing security tools can be relied upon to protect the cluster. This also helps to highlight the strengths and weaknesses of each tool and offers guidance on which scanners are best suited for Kubernetes security auditing based on different use cases.

2.3 Security Threats Classification

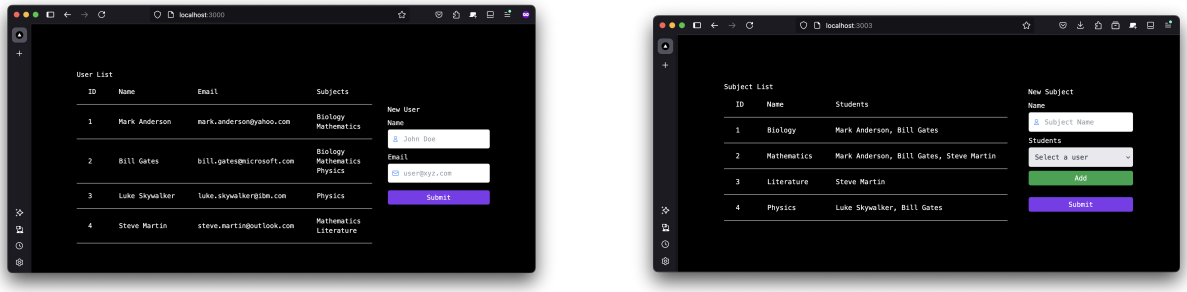
We propose the following classification of the potential Kubernetes security risks. This is largely based on the security best practices gathered by Aquasec [1, 20]. We group related security threats into common subcategories and convert everything into the table format making it easier to navigate and study.

Category	Subcategory	Description
1. Configuration Vulnerabilities	1.1 Misconfigured RBAC	Incorrectly set roles and permissions can lead to unauthorized access.
	1.2 Pod Security Admission	Missing pod security admission labels on the namespaces can allow privileged containers or insecure configurations.
	1.3 Network Policies	Inadequate network policies can expose services to unauthorized access.
	1.4 Resource Limits	Absence of resource limits (CPU, memory) can lead to resource exhaustion.
	1.5 Preemption Policies	Undefined preemption policies/priorities.
2. Container Vulnerabilities	2.1 Base Image Vulnerabilities	Using container base images with known vulnerabilities, using **latest** tag.
	2.2 Outdated Packages	Containers running outdated software with known exploits.
	2.3 Exposed Secrets	Sensitive data (tokens, passwords) exposed in environment variables or volumes.
3. Kubernetes API Server Vulnerabilities	3.1 API Server Exposure	Unrestricted access to the Kubernetes API server.
	3.2 Audit Logging	Lack of or improperly configured audit logging that hinders incident detection and response.
	3.3 ETCD Data Exposure	Unsecured etcd exposing sensitive cluster data.
4. Network and Communication Vulnerabilities	4.1 Service Exposure	Services unnecessarily exposed to the internet.
	4.2 Ingress/Egress Controls	Inadequate controls over ingress and egress traffic.
5. Runtime and Execution Vulnerabilities	5.1 Runtime Privileges	Containers with root or elevated privileges.
	5.2 Insecure Syscalls	Usage of insecure system calls from within containers.

Table 2.2: Kubernetes security threat classification.

2.4 Experimental Environment

To test the efficiency of the selected tools, a designated Kuberntes environment was created. We have chosen Rancher Desktop for provisioning a local instance of Kubernets. Then, a simple demo application, preemptively exposed with vulnerabilities was deployed inside this cluster. It includes two interconnected microservice apps, simulating a simple solution for user and subject management. Each application is composed of a simple Spring Boot Java application, NextJS frontend and a PostgreSQL database. We have chosen Spring Boot, ReactJS and PostgreSQL as these are the most popular technologies for miscroservice applications nowadays. Business logic of these application is simple and is irrelevant to our purpose. Figure 2.2 capture the user interface of those applications. We made **demo-users** application as insecure as possible, while leaving **demo-subjects** application without any explicit security configuration. In this section we describe the features of the environment, while mapping some of the security threats from the Section 2.3 to their implementation in the configuration files.



(a) **demo-users** application. (b) **demo-subjects** application.

Figure 2.2: Demo applications.

2.4.1 Cluster and Namespaces

For our test we used the most recent Kubernetes version 1.32.0. We leave the Kubernetes API on the default 6443 port and use dockerd as the container engine in Rancher Desktop. Apart from that, no additional configuration is needed for the cluster setup.

We label the **demo-users** namespace with

`pod-security.kubernetes.io/enforce=privileged`,

which enforces the rule that any pod running in that namespace must be configured with elevated privileges. Privileged containers can access the host system's devices, and resources like `/dev` or `/sys` that are normally restricted in regular containers. They may be granted additional Linux capabilities (such as `SYS_ADMIN` or `NET_ADMIN`), allowing the container to perform actions that are typically restricted (e.g., managing network interfaces, mounting file systems, etc.).

2.4.2 Docker Images

To make the images less secure, we have avoided any reecommendations for the Docker image security. That is, for **demo-users** we run the images as root by default. We explicitly expose ports 9229 (NodeJS debug port) and 22 (SSH port) to open them up for a potentional attack. Additionally, we define environment variables containing sensitive information. Finally, we use `latest` image tag, so that the image with undeterministic nde version and potentially unpatched vulnerabilities is used for the build. Listing 2.5 demonstrates the Dockerfile for the **demo-users** frontend image.

```
# use the latest image from Dockerhub
# use image tag instead of hash
FROM node:latest

WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . ./
COPY entrypoint.sh /app/entrypoint.sh
RUN npm run build && \
    chmod +x /app/entrypoint.sh

# sensitive information in environment variables
ENV AUTH_TOKEN=QmVhcmVyIHh3SmRhMDl4T0NhNWFQUXpxM2NjeHF3Vwo=

EXPOSE 3000
# expose unnecessary ports
EXPOSE 9229 22

# run as root
USER root

ENTRYPOINT /app/entrypoint.sh
```

Listing 2.5: **demo-users** frontend image.

2.4.3 Helm Charts

Vast majority of the misconfigurations belong to the Helm charts. We explicitly grant the cluster administrator rights to the ServiceAccount running the **demo-users** Deployment (see Listing 2.6). This means that a potential attacker obtains full cluster

control if he gets access to the pod. Then, we do not restrict or limit the CPU and memory resources available to the pod. Thus, a pod can, potentially, exhaust the node resources, significantly disrupting critical workload availability. We do not define any preemption policies, which makes Kubernetes scheduler treat our workloads as the same priority class. Lastly, we explicitly grant the containers all Linux capabilities, making the root filesystem writable and enforce root user to run the containers. Listing 2.7 provides a snippet from the Helm values, which define some of the security configuration parameters.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: {{ include "demo.serviceAccountName" . }}-rolebinding
subjects:
- kind: ServiceAccount
  name: {{ include "demo.serviceAccountName" . }}
  namespace: {{ .Release.Namespace }}
roleRef:
  kind: ClusterRole
  # 1.1. Misconfigured RBAC
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

Listing 2.6: RoleBinding definition for **demo-users** ServiceAccount, granting it cluster administrator rights.

```
# 6.1 and 6.2
securityContext: {}
  capabilities:
    add:
    - ALL # Grants all capabilities to the container
  # Allows the root filesystem to be writable
  readOnlyRootFilesystem: false
  runAsNonRoot: false # Allows the container to run as root
  runAsUser: 0 # Runs the container as the root user
```

Listing 2.7: **demo-users** Helm Chart values snippet, granting the container unnecessary permissions.

Chapter 3

Implementation

To achieve the most precise results it was necessary to run different scanners multiple times. JSON reports had to be then manually parsed to extract the data on a number of occassions. In order to automate this process and develop an application that would vastly improve the monitoring possibilities inside a Kubernetes cluster, we needed a tool that aggregates multiple scanners into one solution, automatically parses the results and visualizes them. Thus, KSA dashboard has emerged, comrised of the three components, two backend ones and one frontend: **ksa-aggregator**, **ksa-parser** and **ksa-dashboard**, respectively. Since the microservices are developed specifically for the Kubernetes environment and need to be closely integrated with it, Go was chosen as the primary language for the backend microservices. ReactJS was chosen for the frontend as one of the simplest yet very powerful JavaScripts frameworks (though we used Typescript). This chapter focuses on some of the implementation aspects of the KSA Dashboard.

3.1 Architecture and Software Design

Architecturally, the software product is comprised of three components. As shown on the schematic overview of the deployment (Fig 3.2), these components use HTTP API requests and Websockets for the communication. We can safely use unprotected HTTP protocol without TLS as the components are deployed inside one namespace and use internal communication channels only. We use websockets to notify frontend of events on the backend, which makes frontend very responsive, but tightly couples these components. A message queuing system like Kafka or RabbitMQ was considered as an alternative solution that would loosen the components and make them less dependant on each other, but it was decided against it as the system is not very big and messaging does not occur too often. Using message broker would only add extra complexity to the solution in our case.

From the feature perspective, the tool was designed to help DevSecOps engineer manage Kubernetes secuirty and, thus, serves the following requirements:

- User must be able to trigger the scan from the dashboard.
- Scanning and parsing should occur automatically without user interaction.
- Scan results should be persisted in the database for the future use.
- User must be able to browse the scan results.
- User must be able to search through the results.
- User must be able to generate reports based on the scan results.
- User must be able to manipulate findings (delete or mark as resolved).
- Dashboard should provide the possibility to be extended with other security scanners in the future.
- Dashboard should be able to generate graphic visualisation of the scan results.

The component diagram on the Fig 3.1 illustrates the architecture of a Kubernetes-native security scanning system. The process begins with the Dashboard component, which allows users to trigger scans and query results. Scan requests are forwarded to the Aggregator, which continues the process by initiating and watching for the scan job through the Kubernetes API. The Kubernetes API is responsible for scheduling the execution of containerized scanner jobs (such as Trivy, Kube-bench, Prowler and Kubescape) within the cluster. Once the scanners complete their tasks, Aggregator component initiates parsing by sending a request to the Parser component, results are then stored in the Database component, which supports query operations for both the Dashboard and Aggregator. Additionally, the Parser exposes a WebSocket endpoint that enables real-time updates to the Dashboard—notifying it asynchronously when scan results are available. This decoupled, event-driven approach improves system responsiveness and user experience.

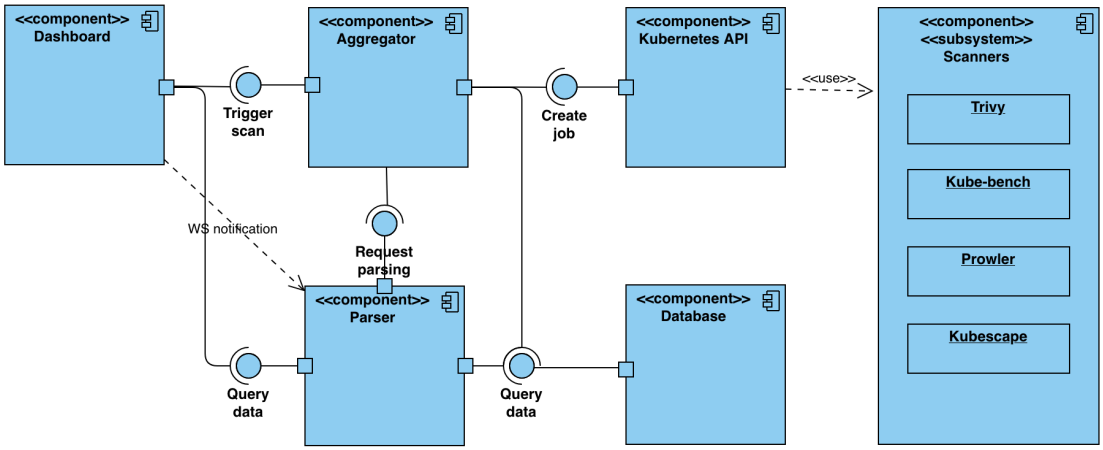


Figure 3.1: Component diagram of the dashboard.

As it can be seen on the Fig 3.2, both Parser and scanner Jobs mount the same PVC, where the results of the scan are stored. Each scanner has its own PersistentVolumeClaim, so that they can perform simultaneous writing uninterrupted. Those PVCs are mounted into specific directories on the Parser pod, that reads reports created by the scan jobs from these directories.

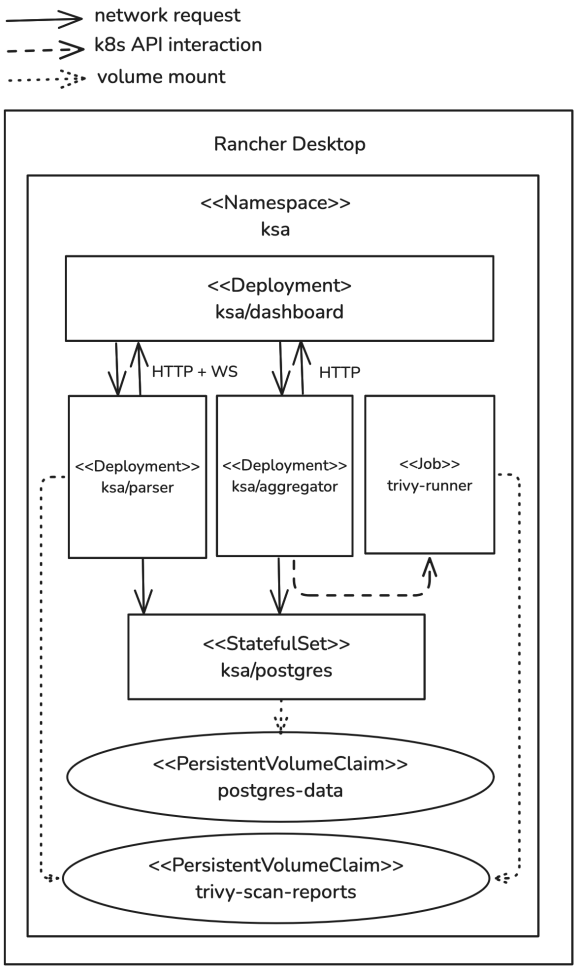


Figure 3.2: A schematic overview of the KSA dashboard deployment.

3.2 Data Design

Data design of the solution is very simple. Table `scanner` holds the supported scanner and its contents are constant, they do not change during runtime. Table `report` is used to store report metadata. Tables `vulnerability` and `misconfiguration` store the findings extracted from the reports. Figure 3.3 depicts those entities and relationshipes between them.

Notice that both **vulnerability** and **misconfiguration** tables have `search_vector` column. This column is generated when a new row is inserted or updated and used

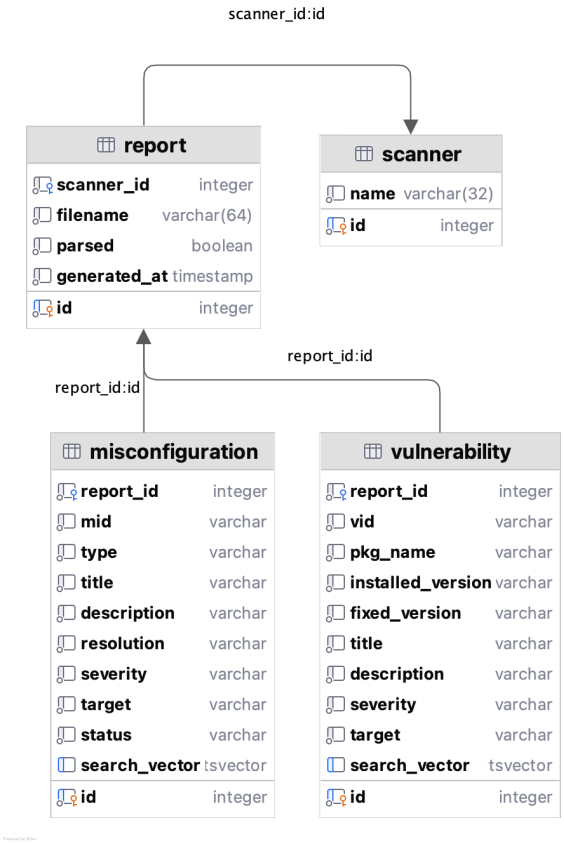


Figure 3.3: Entity relationship diagram visualizing the KSA dashboard database tables and their relationship. Icons next to the column names provide necessary information about the column. Little circle in the bottom left corner depicts `NOT NULL` constraint. Golden key denotes primary key and blue key denotes foreign key. Blue side indicates that this column is indexed.

for the full-text search on these tables. Listing 3.1 shows the definition of the respective triggers for this column, which transform certain fields like misconfiguration id, description and title into tokens, which are then used to perform search like this: `search_vector @@ plainto_tsquery(%s)`, where `%s` is substitute string for the query.

```
alter table misconfiguration
    add column search_vector tsvector;
create index misconfiguration_search_vector_idx on
    misconfiguration using gin (search_vector);

create function misconfiguration_update_search_vector()
    returns trigger as $$
begin
    new.search_vector := to_tsvector('english', new.mid ||
     '␣' || new.type || '␣' || new.title || '␣' ||
```

```
            new.description || ' ' || new.target);
        return new;
    end;
    $$ language plpgsql;

    create trigger misconfiguration_search_vector_trigger
    before insert or update on misconfiguration
    for each row execute function
        misconfiguration_update_search_vector();
```

Listing 3.1: Search vector definition for `misconfiguration` inside the database initial script.

3.3 Parser

To make the product extendable, parser service defines a common interface for all security scanner parsers and provides a common definition of vulerabilty and misconfiguration using Golang structures. Listing 3.2 shows a code snippet of the definition. Golang does not support objects and classes in the traditional sense. Instead, it uses interfaces: any structure that implements the methods of the interface is considered to satisfy that interface.

```
/* Parser defines a common interface
for all security scanner parsers */
type Parser interface {
    Parse(filePath string) ([]Vulnerability,
        []Misconfiguration, error)
    GetResults() interface{}
    GetVulnerabilities() []Vulnerability
    GetMisconfigurations() []Misconfiguration
}
```

Listing 3.2: A common interface for parsers.

We parse the JSON reports using the standard Go module `encoding/json`. However, the structure of those reports varies significantly and sometimes is very complex. Some reports, like the ones produced by Kube-bench for instance, are missing some fields. Kube-bench does not specify the severity of its findings, neither it specifies the target. In this case we have to use default values: we set severity to **HIGH**, for example. Additionally, Kube-bench sets status to **WARN** for manual checks and those have to be remapped onto **MANUAL** to be consistent with the Prowler's notation.

Go's `net/http` library is used to expose a number of endpoints for other services

to use. As the server creates a new goroutine for each incoming HTTP request, we have to be careful with our data. Instances of our parsers are static and shared by those goroutines. That is why each instance has a mutex, that is locked whenever a critical operation is performed on the data and unlocked afterwards.

In order to extend the Parser with a new tool, there are three things to be considered:

1. A new implementation of the parser interface (see Listing 3.2) should be defined. It must have `Parse()` method, which would define parsing process for the scanner.
2. This implementation should be added to the parser initialization inside the main function.
3. Parser should be added to the database.

3.4 Aggregator

Aggregator service defines a common interface for all security scanner runners and provides a common interface for job tracking. Listing 3.3 shows a code snippet of the definition.

```
/* Runner defines a common interface
for all security scanner runners */
type Runner interface {
    Run() error
    GetStatus() JobStatus
    CleanUp() error
    Watch(*sql.DB) (int, string)
}

type JobRunner struct {
    clientset   kubernetes.Interface
    namespace   string
    jobName     string
    scannerName string
    fileName    string
}
```

Listing 3.3: A common inteface for runners.

Aggregator uses Kubernetes API to create and delete jobs and watch for their completion. Each runner is provided with the same Kubernetes clientset. It is a set of generated Go clients that allow us to interact with the Kubernetes API programmatically. It handles authentication, API requests, version negotiation, and resource

management. Runner then defines a job as a Golang struct and sends it to the Kubernetes API for creation.

To extend Aggregator with a new scanner, the following steps should be considered:

1. If the scanner does not provide a Docker image, a custom Docker image should be defined and built.
2. A new Runner implementation should be created. `Run()` method should define and run the Kubernetes job for the scanner.
3. Runner should be initiated inside the main Go function.

3.5 Dashboard

KSA dashboard is, perhaps, the most complex component of all. We are using NextJS framework for the development. The application's code is broken down into 10 custom ReactJS components and defines 8 new types. We mostly use client components, only the API components run on the server side of the NodeJS application. The whole user experience is based on the `useEffect` and `useState` React functions, that reload data from the backend upon changes in the selected filters or search query. Additionally, community-made Lucide React icons are used to enhance user experience.

Figure 3.4 displays the dashboard interface. Top side of the dashboard is reserved for the filters, action buttons and search panel. Upon selecting the scanner and item type (Vulnerability or Misconfiguration), user are able to choose from the available reports to load the items. When "All scanners" is selected, dashboard fetches and displays misconfigurations or vulnerabilites from the latest reports (if such exist) generated by all scanners. First button on the action bar triggers a new scan for the selected scanner. Second button allows to download a JSON report, which includes all of the displayed items. Third and fourth buttons delete or resolve all of the filtered items, respectively. That is, users are able to filter items by a keyword and then delete or resolve all of them at once. This is very useful when, for instance, we are bound by a specific Java version by the contract with the customer. Therefore, we, perhaps, would want to mark all of the vulnerabilites which are filtered by the "Java" keyword as resolved.

The space below the panel is occupied by the list of the security threats. They are collapsed under different categories. Each item can be then also opened to examine the details, such as misconfiguration identifier, type, description, resolution and status. Item titles usually also include the target resource, where the misconfiguration was found.

In the top right corner of the dashboard is a **STATISTICS** button that provides some information on the most recent scan results in graphical format as shown below in Fig 3.5.

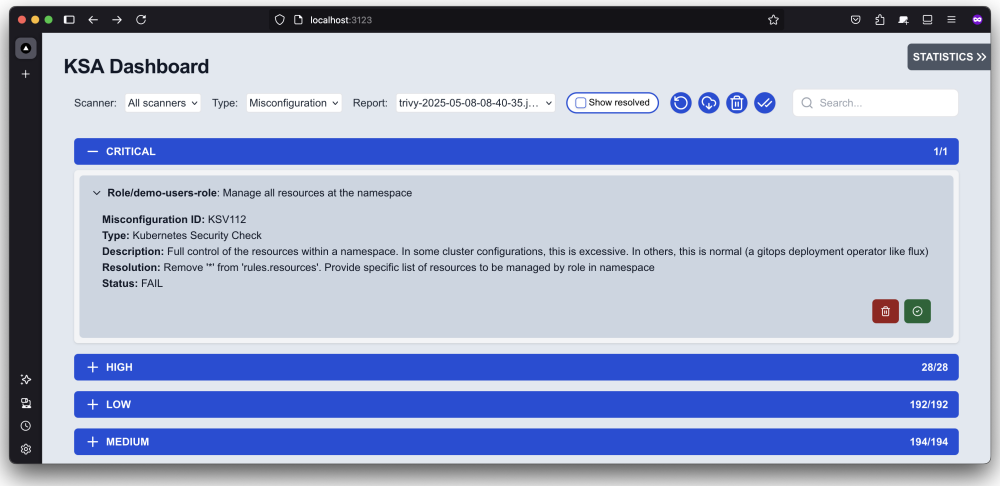


Figure 3.4: User interface of the KSA dashboard.

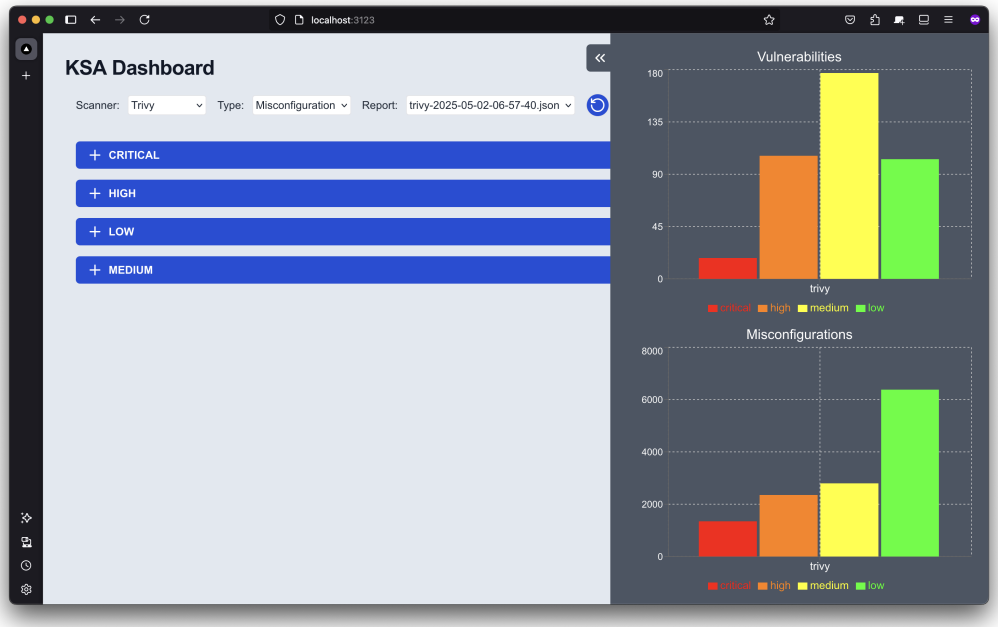


Figure 3.5: Statistics on the KSA dashboard.

To extend the dashboard frontend with a new scanner, no actions are needed as it automatically pulls scanners from the database.

3.6 Build and Deployment

Each of the components of the dashboard has a separate Dockerfile, which allows us to containerize the application. Some of the scanners, which do not have a publicly available Docker image, have to be containerized separately. For instance, we build

Prowler image on a Python base as shown on the Listing 3.4.

```
FROM python:3.12-slim

RUN pip install prowler

CMD ["prowler"]
```

Listing 3.4: Dockerfile definition of Prowler image.

In order to make it easier to build different components of the application, we define a Bash script that does all of the work for us.

Deployment of the whole application is done via the Helm Chart. It defines all of the Kubernetes resources for the application that are deployed inside the cluster. The deployment can be configured using `values.yaml` file, that defines the Helm values, which are then substituted into the Helm templates. Installation is done with

```
helm -n ksa upgrade --install ksa .
```

command, that creates Deployments, StatefulSet, Services and PVCs. There are also `reinstall.sh` and `redeploy.sh` Bash scripts defined to automate the installation and redeployment processes, respectively. Listing 3.5 shows the workloads created by the Helm chart and the Jobs created by the dashboard itself.

When a new scanner is added to the dashboard, the only thing we need to do from the deployment perspective is to define an additional PVC that would hold the reports from the new scanner and mount it to the Parser pod. While it is only a copy-pasting of a few lines, this requires some basic knowledge of Helm. Additional knowledge of Docker would be necessary if an extra image is required for the new scanner.

```
$ k get deployments,sts,jobs,pods

NAME                               READY UP-TO-DATE AVAILABLE AGE
deployment.apps/ksa-aggregator 1/1   1          1         2d21h
deployment.apps/ksa-dashboard  1/1   1          1         2d21h
deployment.apps/ksa-parser     1/1   1          1         2d21h

NAME                          READY AGE
statefulset.apps/postgres 1/1   2d21h

NAME                           STATUS   COMPLETIONS DURATION AGE
job.batch/kube-bench-runner Complete 1/1          44s      2d21h
job.batch/prowler-runner    Complete 1/1          4s       2d21h
job.batch/trivy-runner      Complete 1/1          3m20s    2d11h
```

```
NAME                                    READY STATUS    RESTARTS AGE
pod/ksa-aggregator-7d79b846d5-xzpgt 1/1   Running   0        2d21h
pod/ksa-dashboard-5bd4767cdd-q7ksg  1/1   Running   0        2d21h
pod/ksa-parser-79b55cdf7d-npdjp     1/1   Running   0        2d21h
pod/kube-bench-runner-948bd         0/1   Completed 0        2d21h
pod/postgres-0                      1/1   Running   0        2d21h
pod/prowler-runner-rpdn4            0/1   Completed 0        2d21h
pod/trivy-runner-qp5dw              0/1   Completed 0        2d11h
```

Listing 3.5: Kubernetes workloads for the KSA Dashboard.

Chapter 4

Results

As described in Section 2.2, a set of experiments was performed on the Kubernetes environment with a variety of pre-planted security misconfigurations. Since Kubernetes API Server Vulnerabilities from the Tab 2.2 cannot be implemented in the local Rancher Desktop instance due to the Rancher Desktop's limited cluster configuration options, Tab 4.1 provides only data declared by the scanner documentation in this row.

If we look at the Trivy column, we can see that Trivy does not check for missing Pod Security Admission labels on the namespaces, which can be configured to reject pods with policy violations. Then, Trivy does not check for the missing Network Policies. According to the Aqua Vulnerability database [21], such check exists, but only for clusters deployed using Microsoft Azure services. Furthermore, Trivy does not check for missing of misconfigured Pod Preemption Policies, which bear a risk, according to the Kuberntes documentation [16]: "a malicious user could create Pods at the highest possible priorities, causing other Pods to be evicted/not get scheduled." Trivy shows good performance in search for container vulnerabilities. It has detected the use of **latest** tag and found over a thousand package vulnerabilities inside the containers. However, it did not detect an exposed secret inside the container when executed in cluster scan mode. A separate execution specifically against the image registry has detected the secret. Trivy does not perform any networking checks, unnecessarily exposed services via LoadBalancer did not produce any warnings or failures, but binding of the container port to the host port on the Kubernetes Load Balancer workload has been detected. When it comes to the elevated pod privileges, Trivy was able to detect containers running as root and pods with added capabilties.

Kube-bench performs a pure CIS Kubernetes Benchmark scanning and, thus, does not include any additional checks for missing policies or resource limits. It is not able to perform container image scanning and, therefore, lacks any vulnerability output. Most of the scan results also require manual confirmation. That is, of 59 performed checks only 18 were automated. Moreover, Kube-bench was not able to perform any

Control Plane or etcd benchmarks. It has recognized the only Rancher Desktop node as a Woker Node, thus, only executing Worker Node and Policies benchmarks.

Kubescape has yielded the best results overall. It is the only scanner from the selection that has checks for missing Pod Security Admission and Network policies. It is also the only scanner able to detect misconfigured ingress and egress controls. However, user has to explicitly include those into scanning process as these checks are not performed by the default scan.

Prowler does a great job in looking for misconfigured RBAC, there are 338 checks that detect vulnerabilities in RBAC resources. However, Prowler has no checks for resource limits or any securirity policies. It also cannot scan for any container vulnerabilities or misconfigured network policies. Prowler has a check for secrets as environment variables, but it did not recognize our pre-planted exposed secret as a misconfiguration. There are 9 manual checks for critical file ownership and auditing of the cluster roles.

Table 4.1 shows that all selected tools lack security coverage in some areas. None of the selected tools are able to detect missing or misconfigured Pod Preemption policies or detect exposed secrets in container images from a cluster scan. The former allows Kubernetes to schedule privileged pods and is of minor importance as scheduled privileged pods are still detected by the scanners. The latter, however, bears a major risk of becoming a critical vulnerability opening up the infrastructure for a potential attack. On the other hand, most of the modern container regisrties include a vulnerability scanner, which should be able to detect such threats.

Kubescape detects most of the security threats in different categories. Unfortunately, Kubescape does not yet support control extension with custom frameworks. There is some degree of customization as the user can extend the existing frameworks with custom checks, which he "builds" from the limited amount of predefined configuration parameters. That is, it is currently not possible to extend Kubescape's capabilities to make it scan for missing pod preemption policies or exposed secrets.

Category	Subcategory	Trivy	Kube-bench	Kube-scape	Prowler
1. Configuration Vulnerabilities	1.1 Misconfigured RBAC	yes	yes	yes	yes
	1.2 Pod Security Admission	no	no	yes	no
	1.3 Network Policies	no	no	yes	no
	1.4 Resource Limits	yes	no	yes	no
	1.5 Preemption Policies	no	no	no	no
2. Container Vulnerabilities	2.1 Base Image Vulnerabilities	yes	no	yes	no
	2.2 Outdated Packages	yes	no	yes	no
	2.3 Exposed Secrets	yes[1]	no	no	no
3. Kubernetes API Server Vulnerabilities	3.1 API Server Exposure	yes[2]	yes[2]	yes[2]	no
	3.2 Audit Logging	yes[2]	yes[2]	yes[2]	no
	3.3 ETCD Data Exposure	yes[2]	yes[2]	yes[2]	no
4. Network and Communication Vulnerabilities	4.1 Service Exposure	yes[3]	no	yes	no
	4.2 Ingress/Egress Controls	no	no	yes	no
5. Runtime and Execution Vulnerabilities	5.1 Runtime Privileges	yes	yes	yes	no
	5.2 Containers with root or elevated privileges.	yes	yes	yes	yes

Table 4.1: Secuirty threats detected by the scanners.

Trivy breaks down its findings into four severity categories. Figure 4.1 shows the number of misconfigurations found by the Trivy in each category. Majority of our

[1]Cannot be detected in cluster scan mode.

[2]Declared but not tested.

[3]Trivy does not recognize LoadBalancer service type as misconfiguration, but detects binding of the host ports instead, thus, indirectly detecting service exposure.

pre-planted misconfigurations have fallen into lower categories. Our custom Role with access to all resouces has been recognized as the only **CRITICAL** misconfiguration. Hence, by default configuration Kubernetes has a number of **HIGH** security threats. Writable root filesystem and default security context were detected by the Trivy in this category. Additionally, it has detected binding of container ports to the host ports on the node. However, the containers are the Kubernetes Load Balancer workloads and they must bind host ports in order to expose the service. Therefore, this result is a false positive.

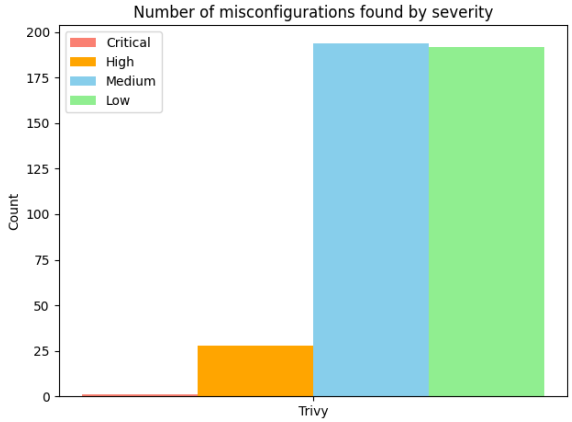
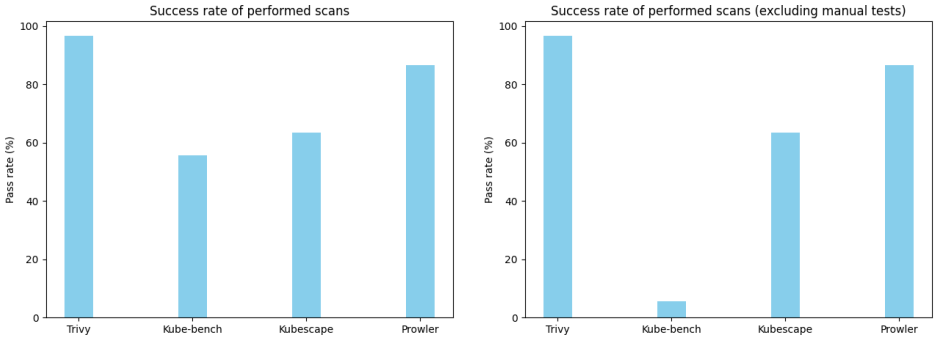


Figure 4.1: Trivy's misconfiguration scan performance.

Figure 4.2 displays the percentage of successfully passed tests executed by each scanner. On the left side we can see the results with manual checks, the right side shows the results excluding the manual tests. As we can see, most of the automated checks performed by the Kube-bench have failed. This is due to the Kube-bench not separating the results by the target. Even if one of the RBAC configuration files is invalid, the whole test is marked as failed, whereas other scanners distinguish between different targets and provide results for them separately.

Kubescape has a broader coverage between different domains of misconfigurations than Prowler or Trivy. Therefore, it has a lower success rate as many of those misconfigurations not covered by the other scanners have been detected as failures by Kubescape. Even though we have discovered a few false positives in Trivy results, it has the highest overall success rate. This, however, does not mean that it has missed some important misconfigurations. Looking at the Fig 4.3, we see that Trivy scan includes a lot of checks that do not fail on the default configuration and distinguishes between all of the target resources.

Trivy's scan did over than 12000 checks on our small demo cluster. Trivy has the most thorough ruleset, which checks for singular misconfigurations. For instance, while most of the scanners check for added pod capabilities in one rule, Trivy includes



(a) Success rate of the tests, including those manually performed.

(b) Success rate of the tests, excluding manually performed tests.

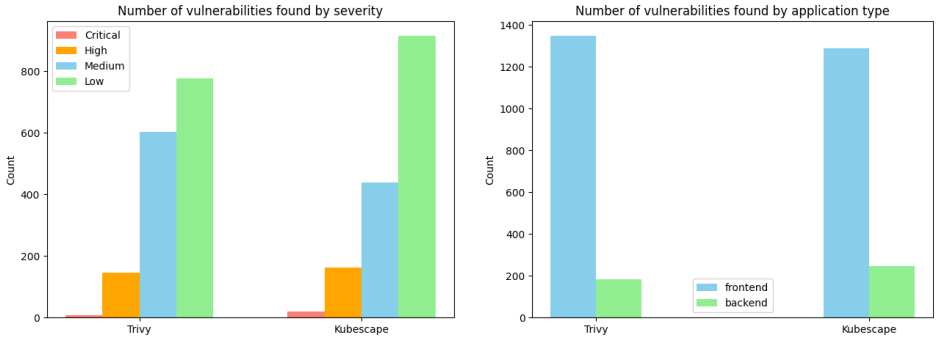
Figure 4.2: Success rate of different scanners, showing the percentage of successfully passed tests.

a separate rule for each added capability.

Trivy and Kubescape show similar results in vulnerability scanning. Kubescape has detected a few additional vulnerbilities that were not detected by Trivy in low severity category. Looking at the Fig 4.5b, we can see that Trivy performs better in our frontend NodeJS applications, where it has found slightly more vulnerabilities. Nevertheless, Kubescape has been able to detect a few critical vulnerabilities in our backend applications, that were completely missed by the Trivy. Upon further investigation we discover that **tomcat-embed-core**, **tomcat-embed-el** and **tomcat-embed-websocket** packages have 13 critical vulnerabilites and yet none of them were detected by the Trivy.

Figure 4.3: Quantity of executed tests by each scanner.

Based on these results we can conclude that Kubescape would be the best choice for a cloud security engineer. It has a broad feature set, which includes both vulnerability



(a) Comparison of vulnerability count by severity.

(b) Comparison of vulnerability count by application type.

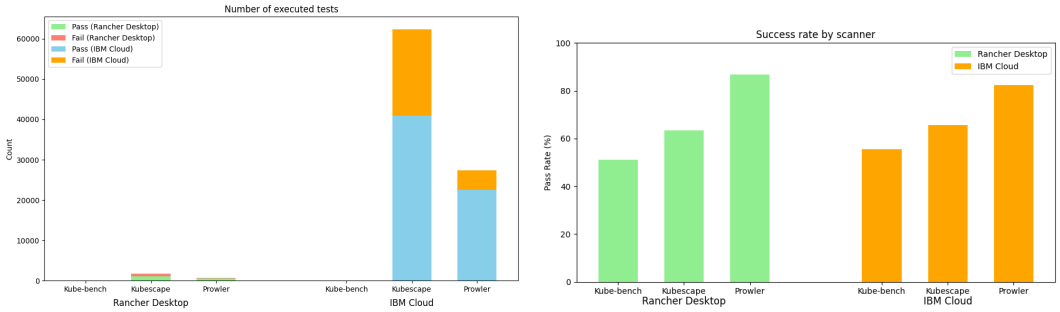
Figure 4.4: Number of vulnerabilites found in demo applications by Trivy and Kubescape.

and misconfiguration scanning. It shows the best performance in vulnerability scanning and was able to detect most of our pre-planted misconfigurations. Since it is not extendable, our recommendation would be to use it in conjuction with other security tool that is able to detect missing preemption policy labels. There were no such tool among those selected for the research, unfortunately. None of the tools could detect an exposed secret inside the container when executed in a cluster scan mode, but this threat should be detected on the build step of the CI/CD pipeline or at the image registry level.

The assessment was also carried out on the enterprise-level cluster within one of the IBM projects. Upon agreement with the customer, we can only publish some of the security threats. The cluster has over 125 namespaces with more than 2000 workloads. Trivy was unable to finish neither of the full-cluster scans, it crashed with timeout exception after several hours of execution. When executed against a single namespace, it produced a report after two hours and detected 19 misconfigurations and over 7000 vulnerabilites inside the containers. Most probably, Trivy does a thorough container image scan, which severely affects its performance. Kubescape managed to finish the scan in a few minutes, detecting over 20000 misconfigurations. Prowler has detected about 5000 misconfigurations in that cluster. Among those were added capabilities for logging agents and IBM proxy workloads, network sharing and privileged containers on the master node proxies and CSI volume drivers. The aforementioned threats, though present an actual risk, can be neglected as they are unavoidable for those workloads. They account for about 70% of those misconfigurations and we would expect them to be ignored when official support for IBM Cloud is added. Among the business workloads there were containers with root privileges and containers with missing security profiles, none other misconfigurations were found there. Kube-bench CIS Benchmark checks have detected 9 failures: `cluster-admin` role usage, default

service account usage, mounting of service account tokens, create pods access, wildcards in Roles and ClusterRoles, access to the secrets, incorrect permissons on the 3 node configuration files.

Figure 4.5 exhibits the results obtained on the IBM Cloud cluster in comparison with locally produced ones. Even though the number of executed tests is significantly higher in the IBM Cloud cluster, pass rates do not vary significantly between Rancher Desktop and IBM Cloud results. Thus, we can assume, that IBM Cloud cluster instance is predisposed with similar misconfigurations as our local cluster, but on a larger scale.



(a) Comparison by the number of misconfigurations found.

(b) Comparison of pass rates for the selected scanners.

Figure 4.5: IBM Cloud results compared with Rancher Desktop.

Finally, as we discovered that Kubescape cannot be extended and there are only two missing checks, we decided to not create a framework of our own, but to create a tool that would aggregate multiple scanners covering the whole range of security domains. A cloud solution for automatic aggregation and parsing of the Kubernetes scanner results has been developed. It is able to parse the reports of multiple scanners and present the results in structured and easily managable format in form of a security dashboard. This dashboard is implemented in a way that would allow further extension with the new scanners. Scan jobs can be directly triggered via the dashboard and the parsing is triggered automatically as the job finishes. Users are able to review misconfigurations and vulnerabilities found by the scanners, search through them, mark failed checks as resolved, remove unwanted items from the results and generate JSON reports. This solution is designed for a DevSecOps specialist and significantly improves his work efficiency by automating multiple processes.

In the future it would be necessary to integrate Kubescape Operator into the dashboard deployment, so that full potential of Kubescape can be achieved. Without the Operator Kubescape can only perform a handful of controls. Integrating AI into the dashboard would bring a lot of benefits: we could filter out duplicates produced by different scanners and generate summaries with the most critical findings. At this moment, duplicates are hard to recognize as different scanners use different internal

ids for misconfigurations and classify them differently. But AI could, perhaps, detect duplciates based on the combination of properties. Automatic failure mitigation is a controversal subject. Usually, automatic configuration changes are not desired, but with proper caution and verification this could be accepted. Therefore, this could be a potential extension of the dashboard. Less far-fetched feature is the generation of PDF or CSV reports. This can be implemented in such a way that user can choose in which format to download the report. Lastly, "Statistics" tab can be reworked into a separate page and additional charts could be added there. The most interesting of them would be the chart that showcases the trent in misconfiguration or vulnerability count over the time based on the generated reports.

Conclusion

A selection of security scanners were executed against a dedicated Kubernetes test environment pre-planted with vulnerabilities. We exclusively selected free and open source tools that are designated for the Kubernetes and are actively supported to this day. Scan results of those tools were analysed and compared. Upon comparison none of the tools were able to detect all of the security threats that we have prepared. However, Kubescape achieved the best result recognizing all but two misconfigurations. It has no ability to detect exposed secrets inside the container nor can it detect missing or misconfigured pod preemption policies. Trivy was also able to find the majority of misconfigurations, but it lacks checks in networking and policies domains. Kube-bench and Prowler showed distinctly bad performance and were able to detect only a few threats.

Kubescape would be the recommended tool for the best security coverage. At this moment, however, Kubescape cannot be extended with custom user-defined checks. Therefore, it would be better to use Kubescape in conjuction with another security scanner, which would complement the results. To aggregate multiple scanners and automate scanning and parsing processes we have developed an extendable Kubernetes security dashboard. The dashboard integrates directly with the Kubernetes API and allows users to trigger scan jobs and browse through their findings. Additionally, users are able to search through the findings, manage them and generate reports. These features make it very useful for the DevSecOps specialist at his job in securing the infrastructure.

The dashboard can be further extended with such features as:

- AI-generated reports, that highlight the most dangerous findings,
- filtering out of the duplicated results from the different scanners using AI,
- charts illustrating the trend in vulnerability counts over time,
- generation of reports in other formats (PDF, CSV, text summary),
- itegration with Kubescape Operator,
- automatic failure mitigation.

Bibliography

[1] Aquasec kubernetes vulnerability database - misconfigurations. https://avd.aquasec.com/misconfig/kubernetes/, Accessed: 7.3.2025.

[2] Kubescape github page. https://github.com/kubescape/kubescape, Accessed: 20.4.2025.

[3] Prowler github page. https://github.com/prowler-cloud/prowler, Accessed: 20.4.2025.

[4] Regolibrary github page. https://github.com/kubescape/regolibrary, Accessed: 20.4.2025.

[5] Trivy github page. https://github.com/aquasecurity/trivy, Accessed: 20.4.2025.

[6] Trivy github page. https://github.com/aquasecurity/kube-bench, Accessed: 20.4.2025.

[7] National Security Agency, Cybersecurity, and Infrastructure Security Agency. *Kubernetes Hardening Guide v1.2*. NSA, 2022. PDF available online at https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF.

[8] Moez Ali. Kubernetes vs docker: Differences every developer should know. https://www.datacamp.com/blog/kubernetes-vs-docker, Accessed: 16.12.2024.

[9] D. B. Bose, A. Rahman, and S. I. Shamim. 'under-reported' security defects in kubernetes manifests. In *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*. IEEE, 2021.

[10] Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. Learning state machines to monitor and detect anomalies on a kubernetes cluster. In *17th International Conference on Availability, Reliability and Security (ARES '22)*, New York, NY, USA, 2022. Association for Computing Machinery.

[11] D. A. Castillo Rivas and D. Guamán. Performance and security evaluation in microservices architecture using open source containers. In *Communications in Computer and Information Science*, volume 1388. Springer, 2021.

[12] The MITRE Corporation. Containers matrix. https://attack.mitre.org/matrices/enterprise/containers/, Accessed: 20.4.2024.

[13] Center for Internet Security. *CIS Kubernetes Benchmark v1.11.0 - 04-01-2025*. CIS, 2025. PDF available online at https://downloads.cisecurity.org/.

[14] The Linux Foundation. Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/, Accessed: 16.12.2024.

[15] The Linux Foundation. Kubernetes official webpage. https://kubernetes.io/, Accessed: 4.11.2024.

[16] The Linux Foundation. Pod priority and preemption. https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/, Accessed: 20.4.2025.

[17] SUSE Rancher Engineering group. Rancher desktop by suse. https://rancherdesktop.io/, Accessed: 2.12.2024.

[18] Marko Lukša. *Kubernetes in Action*. Manning Publications Co., 20 Baldwin Road, PO Box 761, Shelter Island, NY 11964, 2018.

[19] V. B. Mahajan and S. B. Mane. Detection, analysis and countermeasures for container based misconfiguration using docker and kubernetes. In *2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS)*. IEEE, 2022.

[20] Rani Osnat. Kubernetes security basics and 10 essential best practices. https://www.aquasec.com/cloud-native-academy/kubernetes-in-production/kubernetes-security-best-practices-10-steps-to-securing-k8s/, Accessed: 7.3.2025.

[21] Aqua Security. Aqua vulnerability database. https://avd.aquasec.com/misconfig/kubernetes, Accessed: 20.4.2025.

[22] Stephanie Susnjara and Ian Smalley. What is containerization? https://www.ibm.com/topics/containerization, Accessed: 4.11.2024.

[23] Stephanie Susnjara and Ian Smalley. What is kubernetes? https://www.ibm.com/topics/kubernetes, Accessed: 4.11.2024.

[24] Prowler Team. Prowler open source documentation. https://docs.prowler.com/projects/prowler-open-source/en/latest/, Accessed: 20.4.2025.

Appendix

Source code and configuration files are available as an attachment to this paper and can be downloaded online from the Github repository located at

https://github.com/pavel-semenov-1/kubernetes-security-assessment.

Repository is structured as follows:

- **artifacts** folder contains some of the reports from the scanners,
- **docker** folder contains the source codes and Dockerfile definitions for all applications,
- **docs** folder has some documentation on the initial experiments,
- **k8s** folder contains Helm templates for all applications,
- **latex** folder contains LaTeX sources of this paper,
- **web** folder holds the source code the thesis webpage.

There are multiple **README.md** files, which should guide anyone willing to recreate our test environment or deploy the KSA Dashboard.