Hardening Kubernetes Clusters

Reducing Attack Surface in Kubernetes by means of Rootless Containers, Network Policies and Role Based Access Control

Bachelor Thesis

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science in Engineering

to the University of Applied Sciences FH Campus Wien
Bachelor Degree Program: Computer Science and Digital Communications

Author:

Guntram Björn Klaus

Student identification number:

c2110475170

Supervisor:

BSc. MSc. Bernhard Taufner

Date:

dd.mm.yyyy

Declaration of authorship:

I declare that this Bachelor Thesis has been written by myself. I have not used any other than the listed sources, nor have I received any unauthorized help.

I hereby certify that I have not submitted this Bachelor Thesis in any form (to a reviewer for assessment) either in Austria or abroad.

Furthermore, I assure that the (printed and electronic) copies I have submitted are identical.

Date: Signature:

Abstract

(E.g. "This thesis investigates...")

Kurzfassung

(Z.B. “Diese Arbeit untersucht...”)

List of Abbreviations

DNS	Domain Name System
CPU	Central Processing Unit
RAM	Random Access Memory
OS	Operating System
API	Application Programmable Interface
REST	Representational State Transfer

Key Terms

Container
Kubernetes
Cloud
Security

Contents

1 Introduction

1.1 Enterprises and Cloud

Over the last years, there has been a significant shift in the way enterprises around the world deploy, scale and maintain the lifecycle of their computing infrastructure and the software that runs on top of it. Applications used to be primarily constructed as one large software unit that bundles all features, business logic, the user interface and data access components. Increasing necessity for scalability, flexibility and maintainability made organizations transition from such monolithic architectures to Microservices, which are smaller, separated but loosely coupled software units implementing one component of the larger system at hand.

This shift goes hand in hand with 'DevOps', a philosophy and an approach to the software lifecycle which aims to increase the efficiency of an organization's by emphasizing integration of responsibilities between development and operations. With the release of Docker in 2013 and Kubernetes in 2014, companies have increasingly adopted microservice architecture.

Kubernetes has become the go-to choice for managing containerized applications, particularly in cloud-native environments. Its ability to automate tasks, scale applications, and support a wide range of use cases makes it a powerful tool for both large enterprises and small teams.

1.2 Research Objectives

1.3 Methodology

1.4 Structure

2 Concepts

2.1 Containervirtualization

The high level definition of a container is that of a lightweight, standalone, and executable software package that includes everything needed to run a piece of software. A container bundles the code itself, any required runtime environments, dependencies, settings as well as system tools and libraries into a tangible, portable format, which is executed in an isolated environment on the underlying host machine. To exemplify this definition, picture a containerized enterprise Java application. Such a container specifies the underlying Host-OS, the JDK, which includes the JVM and other Java binaries, an application server such as JBoss, Wildfly or Tomcat, the source code itself with its dependent libraries and configuration files, and host packages such as bash, curl, jq, envsubst, etc. This isolation of environments is possible through the namespace and cgroup technology of the Linux kernel

2.1.1 Linux Kernel

The kernel: The operating system kernel is the core component of any operating system. It acts as a communication bridge between a user's applications and the underlying hardware. It is responsible for managing the fundamental functionality of the system, from scheduling processes to providing resources to applications.

The kernel performs a number of crucial functions, including low-level networking, disk storage, and the control of processes and memory. The primary goal's of the kernel include:
1) Establish the process that will run next on the CPU, when it will run, and for how long.
2) Keep track of the amount of memory being used of each process
3) Act as mediator between processes and hardware
4) Handle requests for execution from the processes through system calls
The kernel performs its tasks in its own, designated part of the system memory called "kernel space". In contrast, custom applications such as web-browsers, texteditors and mail clients run in "user space". In order to safeguard memory and hardware against harmful or abnormal program behaviour, kernel space and user space are kept apart. While the kernel has access to all of the memory, processes operating in the user space can only access a portion of it. Additionally, processes that are operating in user space are not able to access kernel space. Only a limited portion of the kernel is accessible to user space processes through an interface that the kernel exposes, the system calls. Through system calls, a user-level program invokes services from the kernel.

The Linux kernel offers a wide variety of features which can be consumed by programmes. The isolated, virtualized environements which make up so called containers are mainly achieved by the namespace and cgroup technology of the Linux kernel.

2.1.2 Linux Control Groups

Cgroups, short for control groups, are a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, etc.) of a collection of processes.

Cgroups are a key component of containers because there are often multiple processes running in a container that you need to control together. In a Kubernetes environment, cgroups can be used to implement resource requests and limits and corresponding QoS classes at the pod level.

Control Groups (cgroups) are a Linux kernel feature that provides hierarchical partitioning of system resources among sets of processes. They are used to isolate and control the resource usage of individual processes or groups of processes. Resource Quotas cgroups allow for limiting the resource consumption of processes or groups of processes. This includes limiting CPU, memory, I/O, and network bandwidth usage. These limits are enforced by the kernel, and they ensure that processes do not consume excessive resources and that other processes can access the resources they need. Resource Monitoring cgroups provide detailed resource usage statistics for processes or groups of processes. This information can be used to identify resource bottlenecks, troubleshoot performance issues, and optimize resource allocation. cgroups also provide a mechanism for collecting and exporting this data to other tools, such as monitoring systems and performance analysis tools. Isolation cgroups isolate processes or groups of processes from each other, preventing resource contention and ensuring fair resource allocation. This is important for preventing one process from monopolizing resources and starving other processes. Hierarchical Grouping cgroups support hierarchical grouping of processes, allowing for flexible resource management based on application requirements and user-defined policies. This means that you can create nested cgroup hierarchies that represent the different levels of a system or application. For example, you could have a cgroup hierarchy for each user, each application, or each container. Integration with Docker cgroups are used by Docker to isolate and manage containers. This ensures that each container has its own dedicated resource pool, which prevents containers from interfering with each other.

2.1.3 Linux Namespaces

Namespaces are Linux kernel mechanisms that provide a hierarchical partitioning of the system environment, allowing processes to operate within their own isolated virtualized environment. Each namespace provides a separate view of the system, effectively hiding the existence of other namespaces and their contents. Types of Namespaces Linux provides five main types of namespaces: Process Namespace: Each process within a container has its own process namespace, separating it from other processes in the host system. This prevents processes from directly accessing or interfering with each other's resources. Network Namespace: Each container has its own network namespace, allowing it to have its own private network stack and IP address. This enables containers to communicate with each other without affecting the host network. Mount Namespace: Each container has its own mount namespace, allowing it to have its own view of the filesystem hierarchy. This prevents containers from accessing the host filesystem and allows them to have their own isolated root filesystem. UTS Namespace: Each container has its own UTS namespace, which manages the system's hostname and domain name. This allows containers to have their own unique hostname and domain name, preventing conflicts with other containers. IPC Namespace: Each container has its own IPC namespace, which manages inter-process communication (IPC) mechanisms like shared memory, semaphores, and message queues. This prevents containers from interfering with each other's IPC mechanisms.

2.1.4 Container Images and Runtimes

Finally: Piecing together the functionality of cgroups and namespaces: screenshot of Companies like Docker, and initiatives such as the "Open Container Interface" build sophisticated products and standards around these features.

A container image is a snapshot of a filesystem that contains the application's code, runtime, and dependencies. It encapsulates the entire environment required to run the application, including the operating system, libraries, and any other necessary files. Container images are created using tools like Docker or Buildah, which package the application's source code, dependencies, and environment settings into a single, immutable image.

A container runtime is a software platform that manages the creation, execution, and lifecycle of containers. It provides the underlying infrastructure and mechanisms for running container images, ensuring that they are isolated from each other and the host system. The runtime maps the container's filesystem to the host system's memory, handles network communication, and manages resource allocation for each container. Popular container runtimes include Docker Engine, Podman, and CRI-O.

The relationship between images and runtimes is that images are the blueprints for containers, while runtimes are the engines that bring them to life. The runtime receives a container image, extracts the required components, mounts the image's filesystem, and establishes communication between the container and the host system. During execution, the runtime manages the container's resources, handles network traffic, and ensures the integrity of the container's environment.

2.2 Kubernetes

Kubernetes is an open-source system for for managing containerized workloads and services. It provides a portable, extensible, and scalable platform that automates the deployment, scaling, and management of applications.

2.2.1 Components

A Kubernetes cluster is separated into a control plane and data plane. The control plane groups the components responsible for the underlying system of Kubernetes itself. The data plane hosts the actual user containers. Machines that run control-plane services are referred to as master-nodes, whereas machines that host user applications are referred to as worker-nodes. There are four core control-plane components which work together to manage the cluster and ensure that applications run smoothly. These components run as containers themselves inside the reserved 'kube-system' namespace.

API Server

The API server is the entrance to the Kubernetes cluster. It provides a REST API that allows users and applications to interact with the cluster and manage its resources. The API server is responsible for handling requests from clients, validating them, and enforcing access control policies. It allows you to query the state of the cluster, as well as the configuration of all resources and the status of all pods.

ETCD

Etcd is a distributed key-value store that is used to store the cluster's state. It is highly available and replicated across multiple nodes, ensuring that the cluster's state is always consistent. Etcd is used by all of the control plane components to store information about the cluster, such as the configuration of resources and the status of pods.

Kube Scheduler

The scheduler is responsible for assigning pods to nodes in the cluster. It takes into account the resources available on each node, as well as the resource requirements of each pod, to ensure that pods are placed on nodes where they can run successfully. The scheduler also tries to spread pods across multiple nodes to improve availability and resilience.

Controller Manager

The controller manager is a process that runs on each node in the cluster. It is responsible for monitoring the state of the cluster and taking corrective action when necessary. The controller manager watches for events, such as the creation or deletion of pods, and takes action to ensure that the cluster remains in a healthy state. For example, the controller manager can restart pods that are unhealthy or create new pods to meet demand.

Kube DNS

In addition, Kube-DNS provides DNS resolution for services and pods within the cluster. It creates DNS records for services, allowing pods to reach services by name rather than IP address. This simplifies the process of accessing services and makes it more consistent with traditional application development without the need to map IP addresses to service names. It also acts as the authoritative DNS server for the cluster's default domain, which is 'cluster.local'. This ensures that DNS queries for names within the cluster are resolved correctly and consistently across all pods in the cluster. For example, if a service named 'frontend' inside the 'myapp' namespace exposes an application, it can be called from another application inside the cluster via the domain 'frontend.myapp.svc.cluster.local' or just 'frontend.myapp'.

Kubelet

The Kubelet is an agent process that runs on each node in a Kubernetes cluster. Its primary responsibility is to manage the lifecycle of Kubernetes pods, which are the fundamental unit of deployment in Kubernetes. Kubelet ensures that pods are running, healthy, and up-to-date with the desired state. It also interacts with the Container Runtime Interface (CRI) to run and manage containers, and it communicates with the Kubernetes API server to retrieve and update pod information.

Kube Proxy

Next to the Kubelet, the Kube-proxy, is a network proxy that runs on each node in a Kubernetes cluster. It is responsible for maintaining network connectivity between pods and services. Services are abstractions that represent groups of pods that expose their endpoints to the outside world. Kube-proxy translates service definitions into network rules that direct traffic to the appropriate pods. This ensures that requests to a service are routed to the correct pods, even if the pods are dynamically provisioned or scaled.
Summarizing the aforementioned concepts, figure 3.1 depicts an overview over Kubernetes. ....test

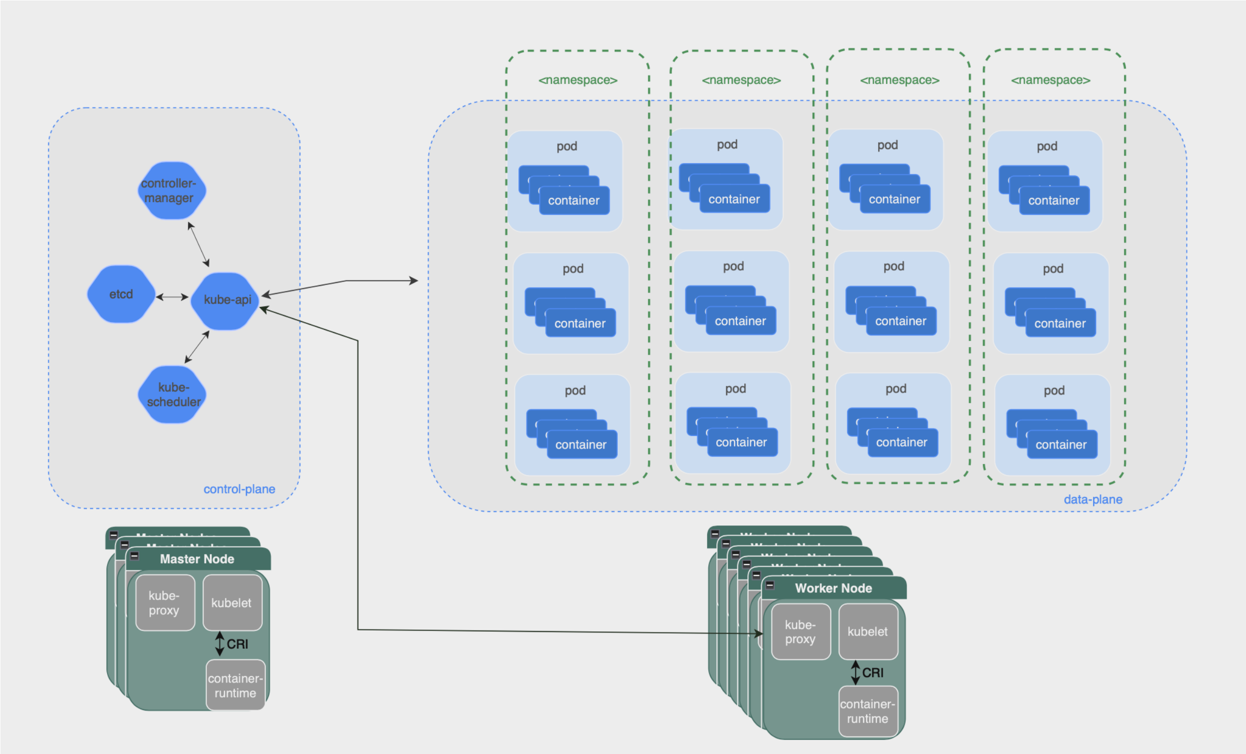

Figure 2.1: Kubernetes Architecture Diagram [source: author]

2.2.2 Cluster Objects

Kubernetes Objects are detailed specifications of the desired state of an application and the cluster as a whole. Objects are formulated in YAML syntax and are communicated to the Kubernetes API using the 'kubectl' command line utility.

Kubernetes Namespaces

Namespaces are virtual partitions in Kubernetes that arrange and segregate resource groupings inside a single cluster. Through namespaces the cluster is logically separated into more manageable, smaller sections. Improved resource management, security distinction, and coordination amongst several teams or projects using a single Kubernetes infrastructure are made possible as a result. Kubernetes objects which define the state of an application, such as 'Deployment', 'Service' and 'ConfigMap' are namespaced. Objects which define the cluster as a whole or which pertain to all applications in all namespaces are not namespaced and exist globally. Namespace isolation prevents unintended interactions or interference between applications, maintaining the integrity of each deployment. Namespaces can be assigned different security policies, allowing for the enforcement of custom authorization rules and access control mechanisms. This helps protect resources and restrict unauthorized access between selected namespaces within the cluster.

Pod

In Kubernetes, pods are the fundamental building block for deploying and managing containerized applications. A pod is a group of one or more containers, which are tightly coupled and share resources such as network namespaces and filesystems. This tight coupling allows pods to be treated as a single unit for scheduling, management, and resource allocation. Pods expose the ports exposed by its containers, allowing external traffic to reach its intendend recipient.

Deployment

labels image replicaset amount of instances

Service

Service In Kubernetes, a Service is an abstraction layer that defines a logical grouping of Pods and exposes them as a single, unified resource to the network. It simplifies the process of accessing and managing Pods by providing a stable endpoint and load balancing capabilities. Load Balancing: Services provide load balancing capabilities, distributing traffic across multiple Pods within the Service. This ensures that requests are handled efficiently and that no single Pod becomes overloaded.

External Exposure: Services can be exposed to external clients, allowing them to access the application running on the Pods. This enables applications to be accessible from outside the Kubernetes cluster, such as from the internet or another network.

Different Service Types: Kubernetes supports different types of Services, each with its own characteristics and capabilities. These include ClusterIP, NodePort, LoadBalancer, and Ingress.

Service Discovery: Services facilitate service discovery, allowing Pods to find and connect to each other within the cluster. This is crucial for distributed applications that communicate with each other.

Health Checks: Services can integrate with health checks to ensure that only healthy Pods are part of the Service's load balancing pool. This prevents traffic from being directed to unhealthy Pods and ensures that the application remains available.

Selectors and Labeling: Services use selectors to identify the Pods that belong to the Service. Selectors can be based on labels applied to the Pods, allowing for flexible and dynamic Service configurations.

DNS Resolution: Services can be resolved to DNS names, simplifying the process for clients to access the application. This provides a consistent and human-readable way to address the application.

ConfigMap

A ConfigMap is a key-value data store that can be mounted into Pods as environment variables, files, or command-line arguments. It provides a way to decouple configuration data from container images, making applications more portable and flexible.

Store Unstructured Data: ConfigMaps store unstructured data, such as configuration files, environment variables, or command-line arguments. This allows for storing a variety of configuration information in a centralized location.

Mountable into Pods: ConfigMaps can be mounted into Pods as environment variables, files, or command-line arguments. This allows applications to access and utilize the configuration data without modifying their container images.

Dynamic Updates: ConfigMaps can be updated dynamically, without requiring Pod restarts. This enables applications to receive configuration changes without downtime, ensuring a consistent experience for users.

Label-based Selection: ConfigMaps can be selected based on labels, allowing for targeted application deployments. This simplifies configuration management and facilitates dynamic updates.

Secret Support: ConfigMaps can also store sensitive data in encrypted form using Kubernetes Secrets. This protects sensitive information and prevents it from being exposed in plain text.

Version History: ConfigMaps track their versions, providing a history of configuration changes. This facilitates rollbacks to previous configurations and simplifies troubleshooting.

Human-readable Names: ConfigMaps have human-readable names, making it easier to identify and manage configuration data.

Ingress

Ingress

In Kubernetes, an Ingress is a controller that manages external traffic routing to a set of Services within the cluster. It acts as a front door for applications, handling load balancing, SSL termination, and other advanced traffic management tasks.

Key Features of Kubernetes Ingresses:

External Traffic Routing: Ingresses direct external traffic to the appropriate Services within the cluster. This allows applications to be accessible from outside the Kubernetes cluster, making them more widely available to users.

Load Balancing: Ingresses handle load balancing across multiple Services, ensuring that requests are distributed evenly and no single Service becomes overloaded. This improves the application's overall responsiveness and scalability.

SSL Termination: Ingresses can terminate SSL/TLS connections, offloading this work from the applications and improving their security posture. This simplifies the application's configuration and reduces the risk of vulnerabilities.

Path-based Routing: Ingresses can route traffic based on the request's path, allowing for different applications or services to be served depending on the URL. This simplifies application deployment and makes it easier to maintain multiple applications.

Customizable Rules: Ingresses can be configured with custom rules that define how traffic is routed, including routing based on headers, query parameters, and more. This provides flexibility in routing traffic to specific applications or services.

TLS Passthrough: Ingresses can support TLS passthrough, which allows for direct communication between clients and the application's Pods without the need for SSL termination at the Ingress level. This is useful for applications that require specific TLS configurations.

Multiple Ingress Controllers: Kubernetes supports multiple Ingress controllers, each with its own strengths and capabilities. This allows for flexibility in selecting the Ingress controller that best suits the application's requirements and infrastructure.

ServiceAccount

n Kubernetes, a ServiceAccount is a security identity that allows pods to access Kubernetes resources and services. It acts as an authentication mechanism for pods, managing their access permissions and controlling their interactions with the cluster. Key Features of Kubernetes ServiceAccounts: Authentication and Authorization: ServiceAccounts provide authentication and authorization for pods, granting them access to specific Kubernetes resources and services. This ensures that pods have the necessary permissions to perform their tasks without granting excessive access. Centralized Identity Management: ServiceAccounts manage identities in a centralized manner, simplifying the process of granting and revoking access permissions for pods. This reduces the complexity of managing access control across multiple pods. Role-Based Access Control (RBAC) Integration: ServiceAccounts can be integrated with Kubernetes RBAC, enabling fine-grained control over pod access permissions. This allows for granular control over what pods can do and what resources they can access. Mounting Secrets: ServiceAccounts can mount Secrets into Pods, providing them with access to sensitive data such as passwords, API keys, or certificates. This simplifies the process of managing sensitive information in Pods. Ownership of Kubernetes Resources: ServiceAccounts can be associated with Kubernetes resources, such as Deployments, Pods, or Jobs, granting them the appropriate ownership and permissions for those resources. This simplifies resource management and enforces ownership rules. Cross-Namespace Access: ServiceAccounts can have access to resources in other namespaces, allowing for secure communication and resource sharing across different namespaces. This promotes flexibility and collaboration in distributed environments.

PersistentVolume

In Kubernetes, a PersistentVolume (PV) is a storage resource that provides persistent storage to Pods. It ensures that data remains accessible even after Pods are terminated and recreated. Key Features of Kubernetes PersistentVolumes: Durability and Reliability: PersistentVolumes provide durable and reliable storage, ensuring that data persists even after Pods are deleted. This is crucial for applications that need to store persistent data. Independent of Pods: PersistentVolumes are independent of Pods, which means that data is not tied to the lifecycle of a particular Pod. This allows Pods to be restarted or replaced without affecting the availability of data. Dynamic Provisioning: Kubernetes supports dynamic provisioning of PersistentVolumes, which means that the cluster can automatically provision storage resources based on the needs of Pods. This simplifies storage management and eliminates the need for manual provisioning. Different Storage Classes: Kubernetes supports different storage classes, which provide different types of storage with varying characteristics, such as performance, durability, and access modes. This allows for flexibility in choosing the appropriate storage for different applications. Multiple PV Providers: Kubernetes supports multiple PV providers, such as cloud-based storage providers, on-premises storage systems, or third-party storage solutions. This enables the use of various storage options and simplifies integration with existing storage infrastructure.

PersistentVolumeClaim

A PersistentVolumeClaim (PVC) is a request for persistent storage from a Kubernetes cluster. It defines the desired storage characteristics and specifications, such as storage capacity, access modes, and storage class requirements.

Demand-driven Storage Allocation: PVCs allow Pods to request storage requirements from the cluster, rather than manually provisioning storage resources. This simplifies storage management and eliminates the need for overprovisioning.

Dynamic Storage Binding: The cluster automatically binds PersistentVolumes to PersistentVolumeClaims based on their compatibility and availability. This simplifies storage provisioning and eliminates manual management.

Guaranteed Storage Allocation: PVCs guarantee that Pods will receive the requested storage capacity, ensuring that applications have the storage they need to function.

Flexible Storage Classes: PVCs can specify the desired storage class, allowing for the selection of different storage types and providers based on application requirements.

Non-blocking Storage Request: PVCs allow Pods to start and run without waiting for the underlying storage to be provisioned. This enables early Pod startup and application availability.

Multiple PVC Providers: Kubernetes supports multiple PVC providers, such as cloud-based storage providers, on-premises storage systems, or third-party storage solutions. This facilitates integration with existing storage infrastructure and enables flexible storage options.

NetworkPolicy

In Kubernetes, a NetworkPolicy is a declarative configuration that defines rules for controlling network traffic between Pods. It allows for fine-grained control over network access, ensuring that Pods can only communicate with authorized sources and destinations.

Key Features of Kubernetes NetworkPolicy:

Fine-grained Network Control: NetworkPolicies allow for granular control over network traffic, specifying which Pods can communicate with each other and with external sources. This enables secure and controlled communication patterns within the cluster.

Policy Enforcement: NetworkPolicy rules are enforced by the Kubernetes network plugin, ensuring that only authorized traffic is allowed to pass between Pods. This protects applications from unauthorized access and malicious traffic.

Label-based Selection: NetworkPolicies can select Pods based on labels, allowing for dynamic and flexible network configuration based on application requirements. This simplifies network management and enables targeted policy enforcement.

Multi-level Visibility: NetworkPolicies provide multi-level visibility into network traffic, allowing administrators to monitor and control network activity at the pod, namespace, and cluster levels. This enables centralized network security and policy enforcement.

Policy Overrides: NetworkPolicies can be overridden at the pod level, allowing for specific Pods to be exempted from the policy rules. This provides flexibility in configuring network access for individual Pods.

Integration with Ingress Controllers: NetworkPolicies can be integrated with Ingress Controllers to control external traffic routing and apply security policies to incoming traffic. This enhances the overall security posture of the application.

Role Based Access Control

RBAC In Kubernetes, Role-Based Access Control (RBAC) is a centralized authorization mechanism that controls which users or groups can access resources and perform specific actions within the cluster. It provides a fine-grained way to manage access permissions and enforce security policies across the Kubernetes environment.

Key Components of Kubernetes RBAC:

Roles: Roles define a set of permissions that can be granted to users or groups. They specify which resources can be accessed, what actions can be performed on those resources, and the scope of the permissions (e.g., namespace-wide or cluster-wide).

ClusterRoles: ClusterRoles are similar to Roles, but they have a broader scope, applying to all namespaces within the cluster. They are typically used to define permissions for system components or users who need access to resources across multiple namespaces.

RoleBindings: RoleBindings associate Roles or ClusterRoles with users or groups, granting them the specified permissions. They define the mapping between identities and the permissions they hold.

Users: Users represent individual identities in the Kubernetes system. They can be created manually or integrated with existing authentication providers, such as LDAP or Active

Directory.

Groups: Groups are collections of users that can be granted permissions together. This simplifies access management for related users and promotes role-based access control.

Namespaces: Namespaces provide a logical separation of resources within the cluster. RBAC policies can be defined within namespaces, allowing granular control over resource access for specific environments or applications.

Policies: Policies are high-level statements that define access control rules for specific resources or actions. They can be expressed in different formats, such as YAML or JSON, and can be integrated with RBAC to enforce fine-grained access control.

3 Literature Review

Kubernetes is highly customizable and offers a range of configuration options which determine the security posture of individual applications and the cluster as a whole. The purpose of this literature review is to explore the security threat landscape of Kubernetes environments. Specifically, recurring concepts and common denominators across vulnerabilities shall be identified and discussed. For this, the database of Common Vulnerabilities and Exposures (CVE), the IEEE database, the ACM digital library and the official Kubernetes feed of CVEs are queried using keywords pertaining to Container and Kubernetes Security. In order to minimize manual labor to research the sheer amount of CVE reports, a Python script was developed to automate the research. The acquired papers, articles and CVE descriptions are skimmed through. The most relevant results are narrowed down and selected for closer inspection. It shall be noted that Kubernetes vulnerabilities do not only entail standard Kubernetes components, but also add-ons deployed on top of 'plain' Kubernetes. Such can be the Nginx Ingress-Controller, a Service-Mesh, CI/CD tools closely embedded into Kubernetes and more. Generally, this can be anything that extends the Kubernetes API through Custom Resource Definitions (CRDs).
CVE: The main goal of the Common Vulnerabilities and Exposures (CVE) Programme is to identify vulnerabilities in a unique way and link particular code bases (such as common libraries and software) to those vulnerabilities. The usage of CVEs guarantees that when discussing or exchanging information about a specific vulnerability, two or more parties can confidently refer to a CVE identifier (CVE numbers). The CVE program defines a vulnerability as:
"A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety)."
In addition, the Common Vulnerability Scoring System (CVSS) "is a method used to supply a qualitative measure of severity"CITE HERE. The three metric groups that make up CVSS are Base, Temporal, and Environmental. Once the Temporal and Environmental metrics have been scored, the Base metrics produce a score between 0 and 10. A vector string, which is a condensed textual representation of the values required to calculate the score, is another way to visualise a CVSS score. Therefore, enterprises, organisations, and governments that require precise and consistent vulnerability severity scores can benefit greatly from using CVSS as a standard measurement system. CVSS is frequently used to prioritise vulnerability mitigation efforts and to determine the severity of vulnerabilities found on a system. All publicly available CVE records are assessed by the National Vulnerability Database (NVD). There are multiple public APIs which allow for querying for various information on CVEs. The goal of the Python script is to quickly collect CVEs for a given keyword with a CVSS base score higher than a given integer. To achieve this, the OpenCVE project's API is queried for all CVEs containing a given keyword. The returned JSON response is filtered for the CVE number. This number is then used to call the NVD API of the NIST organization, whose response can then be filtered for the base severity score according to the CVSS framework. As a result, interesting insights can be gained by programmatic means. The following figure

depicts the collected data on CVE's for the keyword 'Kubernetes'.

Total results: 269
Base score average: 7.13
Base score median: 7.2
Base score modus: 6.5

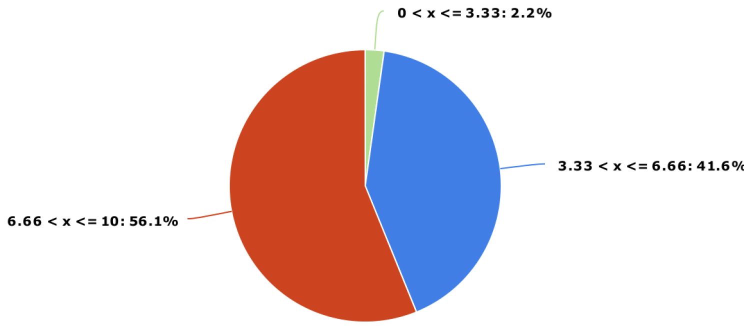

Figure 3.1: Makeup of base scores of CVEs about Kubernetes

The chart below depicts the collected data on CVE's containing either of the keywords 'containerd', 'CRI-O' or 'Docker Container'.

Total results: 75
Base score average: 7.9
Base score median: 7.8
Base score modus: 9.8

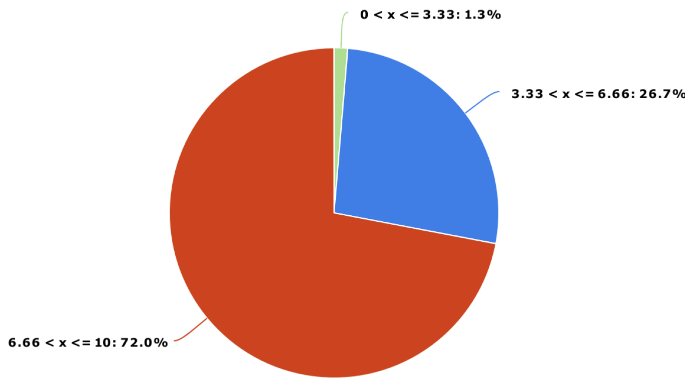

Figure 3.2: Makeup of base scores of CVEs about Kubernetes

Based on collected data, it is noticeable that CVSS base severity scores in the realm of Containervirtualization and Kubernetes tend toward extremes on the higher end. For the keyword 'Kubernetes', more than half of CVEs are rated with a base score higher than 6.66, while there are hardly any scores smaller than 3.33. For CVEs that draw on popular container runtimes, an overwhelming majority close to 75 percent of cases holds base scores between 6.66 and 10. Here, once again, scores smaller than 3.33 are low. These two observations additionally emphasize the necessity for clear security measures when orchestrating containers in Kubernetes. The script enables to quickly retrieve the tendencially more severe vulnerabilities. Some CVEs marked with a CVSS base scores higher than 7 shall be explored in the following chapters.

It is evident that the reliance on Kubernetes comes with the need of a clear security initiative. According to RedHat's report on the state of Kubernetes security of 2022, more than ninety percent of polled organizations underwent at least one security incident in their Kubernetes environment, which, in a third of cases, lead to the loss of revenue or customers. The majority of these incidents were detections of misconfigurations. About a third of respondents reported major vulnerabilities and runtime security incidents in relation to containers and/or Kubernetes which required immediate remediation. In a more recent, similar report conducted by RedHat in 2023, two thirds of respondents had to delay or slow down application deployments because of security concerns. This is a significant increase compared to the 2022 survey, where just over half of participants experienced delays. Three of the most frequently mentioned advantages of containerization include quicker release cycles, quicker bug fixes, and increased flexibility to operate and manage applications. But if security is neglected, you can lose out on containerization's biggest benefit: agility. It becomes apparent that Kubernetes is not something that is installed once and then never looked at again. Rather, a container-based environement that leverages this orchestration technology requires attention for detail and constant, rigurous inspection, despite the great amount of abstraction provided and due to its highly customizable nature. CITE REDHAT 2022 2023
Container Escape
Major vulnerabilities include those which fall under the category of a so called container-escape. Since a container is intended to be a runtime environment isolated from the underlying host, the concept of a container-escape relates to performing an exploit that breaks the confines of exactly this isolation, resulting in full or limited access to the underlying host machine and/or network. A study conducted by Reeves et al. at the end of 2021 investigates the susceptibility of different container runtime systems to escape-exploits by studying a batch of CVE reports. The study identifies three main causes for container escapes. First, mishandled file descriptors, if for example left accessible from within a container under /proc directory, provides malicious actors read and write access to the underlying host filesystem, as seen in CVE-2019-5736. In this reported vulnerability, a container is set up with a symlink from the container's entrypoint to proc/self/exe, which points back to its runC binary, which instantiated the container process. In addition, the container carries a harmful file which is designed to overwrite the file descriptors of any excecuting process that loads it. If an unknowing person executes a binary within the container, which has been manipulated to symlink to /proc/self/exe, the harmful file is able to overwrite the runC binary. The next time another, unrelated container is spawned, it is done by the compromised runC binary. Secondly, missing access control to runtime components could enable adversaries to gain access to UNIX sockets on the host, as reported in CVE-2020-15257. Here, it was possible to connect to the containerd socket, thus enabling actors to issue API commands to freely create new containers on the host, unconstrained by Apparmor, seccomp, or Linux capabilities. Thirdly, under 'adversary-controlled host execution' problems of similar fashion to mishandled file descriptors are mentioned. In this case however, vulnerability exposure starts with host binaries being executed in the container context, which makes it a target for manipulation. In CVE-2019-101(44–47), the shared library "libc.so.6" is altered in such

a way that it mounts the host filesystem when loaded. The new shell loads "libc.so" when the administrator runs "rkt-enter", which is the /bin/bash command by default, to create a new shell in the container. This sets off malicious code embedded in "libc.so", which uses the mknod syscall to construct a block device of the host root filesystem inside the container. As a result, the adversary is able to read and write to the host filesystem.
CVE-2022-0811, which is barely discussed in papers due to its young nature, reports a sophisticated container escape possibility of the CRI-O container engine. Generally, the interface of the Linux kernel accepts parameters which control its behavior. This interface is consumed by CRI-O to set kernel options for a pod. However, the parameter input string is not checked or sanitized, which allows for injecting additional, otherwise undesired parameters. Specifically in the example of CVE-2022-0811, the "kernel.core_pattern" kernel parameter is specified within the parameter of the safe "kernel.shm_rmid_forced" parameter, which controls the kernel's reaction to a core dump. If a core dump is done in a CRI-O container, the parameter states the execution of a malicious binary. This binary sits inside the container but is invoked on the host in the root context of the container from the perspective of the kernel.
CVE-2022-23648 is another vulnerability related to container-escape, this time found in Containerd, a popular Kubernetes runtime. This vulnerability lies in Containerd's CRI plugin that handles OCI image specs containing "Volumes". An attacker can exploit this vulnerability by adding a Volume containing path traversal to the image. This allows them to copy arbitrary files from the host to a container mounted path. More specifically, The vulnerability resides in the "copyExistingContents" function in Containerd's code. This function copies the files from the attacker-controlled volume path to a temporary folder that is later mounted inside a container. An attacker can trick this function into copying arbitrary files from the host filesystem using path traversal. This can lead to the disclosure of confidential information. The severity of this vulnerability is rated as high, with a CVSS base score of 7.5.
CVE-2022-1708 is a vulnerability found in Cri-o, a lightweight container runtime for Kubernetes. This vulnerability is related to the allocation of resources without any limits or throttling, which can lead to uncontrolled resource consumption. The official CVE description states: "The ExecSync request runs commands in a container and logs the output of the command. This output is then read by CRI-O after command execution, and it is read in a manner where the entire file corresponding to the output of the command is read in." It is thus possible to exhaust memory or disk space of the node when CRI-O reads an extensive output of the command. The vulnerability is rated as high, with a CVSS base score of 7.5 and targets system availability.
Cilium is an eBPF-based solution for network connectivity between workloads in Kubernetes CLusters. CVE-2022-29179 reports yet another subjection to the risk of container breakouts. The CVE number is not directly about a technique to gain access outside the container, it rather addresses a vulnerability in Cilium given a successful container escape. It explicitly mentions root containers as a prerequesite. According to this CVE entry, Cilium's service account in versions prior to version 1.9.16 allowed for escalating privileges to those of a cluster admin. This gave adversaries the ability to delete cluster resources such as Pods and Nodes. The entry is marked with a base score of 8.2 which is considered high.
CVE-2020-8554: In the case of CVE-2020-8554, the source is a design defect in the External IPs and Load Balancer IPs components of Kubernetes Services. An application running on a collection of pods can be exposed as a network service in an abstract manner using Kubernetes Services. One or more IPs are used to access a service. When the cluster's nodes are deployed, traffic intended for the service IPs will be routed through one of the backing pods that comprise the service. By assigning IP addresses that are already being used by other

endpoints (internal or external), a malevolent user might intercept all cluster traffic directed towards those IP addresses. With the service shown below, it used to be possible to intercept UDP traffic to IP 8.8.8.8, which is Google's DNS server, and direct it to the "evil-dns-server" pod when it is deployed to the cluster.

```yaml
! service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-evil-service
  namespace: my-evil-namespace
spec:
  selector:
    app: my-evil-dns-server
  ports:
    - name: dns
      protocol: UDP
      port: 53
      targetPort: 9053
  externalIPs:
    - 8.8.8.8
    - 8.8.4.4
```

Figure 3.3: YAML declaration of a malicious service

Of the more recent vulnerabilities with high severity, CVE-2023-5043 and CVE-2023-5044 report problems with the Nginx Ingress Controller. The Nginx Ingress Controller is a popular add-on to Kubernetes with which cluster incoming traffic is managed. Instead of assigning each Kubernetes Service an IP and port directly on the underlying node, the Nginx container serves as a single point of entry and acts as a reverse proxy to the cluster workloads. To expose a cluster workload, a Kubernetes resource of type 'Ingress' is defined. Within this YAML definition, the nginx ingress controller picks up custom configuration through so called annotation snippets. This allows for fine-grained, customized behavior of the controller for a respective service. It was however discovered that specific declarations of the "nginx.ingress.kubernetes.io/configuration-snippet" and "nginx.ingress.kubernetes.io/permanent-redirect" annotations of an Ingress Object can be used to inject arbitrary commands, making it possible to obtain credentials of the said nginx ingress controller. Utilizing this credential, even more cluster secrets could be obtained. Multi-tenant environments are most affected by the issue.

As reported in CVE-2022-24817, Flux2 can reconcile the state of a remote cluster when provided with a kubeconfig with the correct access rights. Kubeconfig files can define commands to be executed to generate on-demand authentication tokens. A malicious user with write access to a Flux source or direct access to the target cluster, could craft a kubeconfig to execute arbitrary code inside the controller's container. In multi-tenancy deployments this can also lead to privilege escalation if the controller's service account has elevated permissions. Within the affected versions range, one of the permissions set below would be required for the vulnerability to be exploited: Direct access to the cluster to create Flux Kustomization or HelmRelease objects and Kubernetes Secrets. Direct access to the cluster to modify existing Kubernetes secrets being used as kubeconfig in existing Flux Kustomization or HelmRelease objects. Direct access to the cluster to modify existing Flux Kustomization or HelmRelease objects and access to create or modify existing Kubernetes secrets. Access rights to make changes to a configured Flux Source. The vulnerability is marked with a CVSS base score of 9.9.
A successful container-escape arguably poses one of the greatest risk in a Kubernetes environment as it provides a starting point from which all three pillars of the CIA triad can

be targeted: Confidentiality, Integrity, and Availability. Once an attacker gains access of the host, secrets can be read, binaries can be altered, network traffic can be inspected and resources can be deleted. In contrast, other vulnerabilities 'only' allow for more restricted exploitability. For example, CVE-2022-1708, CVE——- and CVE———- target the availability of the system, but not the confidentiality or integrity of data. The average CVE scores of
The foundation of such attacks is being able to either freely instantiate containers or freely move and operate from inside a container The most notable ones, which have a severity score above 8, according to the Common Vulnerability
We see that vulnerabilities vary tremendously in their appearance. CVE so and so targets a tool, CVE so and so targets the actual underlying kernel of a k8s node. However, all these things can be prevented or alleviated by applying the concept of least privilege to containers, access to Kubernetes and the Kubernetes network. For example, if a malicious actor cannot freely create files in any desired directory of a compromised container, it significantly reduces his ability to exploit a given vulnerability. Not being able to create that file in the first place, due to missing write permissions, would be a major obstacle for performing a container escape. It is not always possible to completely avoid a vulnerability. This is due to software bugs and unidentified weaknesses in the code that have passed through the testing and review process.

3.1 State of the Art

4 Hardening Measures

These are not problems exclusive to Kubernetes, they exist in any architecture using any tools, setups, services Measures are not exclusive to Kubernetes - containers Generally - regardless of the setup, tools and architecture used, granular access controls, network segmentation or Kubernetes offers an insane range of configuration possibilitites out of the box / native kubernetes or by installing custom tools inside the cluster

4.1 Rootless Containers

Assigning least privileges to a process has always been done The user which executes the application process only has the ability to perform actions on the system that are absolutely required to perform its purpose. The image needs to create a non root user (linux user) which then gets permissions on executables (directory main binary, libs, config files) needed to perform its purpose.

securityContext:

runAsNonRoot true runAsUser 1000 allowPrivilegeEscalation false readonlyrootfilesystem seccomp profile ? selinux options ? this alone does not help if the underlying container image doesnt set up a non-root user. The container wont be able to perform its function

Kyverno and co exist to block the instantiation of non root container in the first place

4.2 Network

Default pod-to-pod network settings, as an example, allow open communication to quickly get a cluster up and running, at the expense of security hardening. Network segmentation..... Default setting is pods are able to communicate freely However it is not necessary for every pod in the cluster to talk to every other pod Also not necessary for pods to have access to the internet Different type of networkpolicy (Calico/Flannel/Cilium use NATTING at node/vm level) NetworkPolicy is a simple declaration of networking capabilities.

4.3 Role Based Access Control

Roles - bindings ClusterRoles - bindings ServiceAccounts

5 Discussion

5.1 Implications for businesses

5.2 Tradeoffs and Difficulties

5.3 Complexity

6 Conclusion

7 Outlook / Future work

Bibliography

List of Figures

List of Tables

Appendix

(Hier können Schaltpläne, Programme usw. eingefügt werden.)