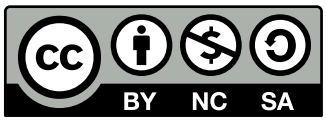
Aalto University
School of Science

Master's programme in Computer, Communication and Information Sciences

Kubernetes inter-pod container isolation

Aarni Halinen

Master's Thesis
2023



Aalto University
School of Science

Author Aarni Halinen

Title Kubernetes inter-pod container isolation

Degree programme Computer, Communication and Information Sciences

Major Computer Science

Supervisor Prof. Mario Di Francesco

Advisors M.Sc. (Tech.) José Luis Martin Navarro, M.Sc. (Tech.) Jacopo Bufalino

Date 1 June 2023 **Number of pages** 43+7 **Language** English

Abstract

The abstract is a short description of the essential contents of the thesis: what was studied and how and what were the main findings. For a Finnish thesis, the abstract should be written in both Finnish and English; for a Swedish thesis, in Swedish and English. The abstracts for English theses written by Finnish or Swedish speakers should be written in English and either in Finnish or in Swedish, depending on the student's language of basic education. Students educated in languages other than Finnish or Swedish write the abstract only in English. Students may include a second or third abstract in their native language, if they wish. The abstract text of this thesis is written on the readable abstract page as well as into the pdf file's metadata via the thesisabstract macro (see the comment in the TeX file). Write here the text that goes into the metadata. The metadata cannot contain special characters, linebreak or paragraph break characters, so these must not be used here. If your abstract does not contain special characters and it does not require paragraphs, you may take advantage of the abstracttext macro (see the comment in the TeX file below). Otherwise, the metadata abstract text must be identical to the text on the abstract page.

Keywords Kubernetes, Container, Docker, Security

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tekijä Aarni Halinen

Työn nimi Opinnäyteen otsikko

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Computer Science

Työn valvoja Prof. Mario Di Francesco

Työn ohjaajat DI José Luis Martin Navarro, DI Jacopo Bufalino

Päivämäärä 1.6.2023 **Sivumäärä** 43+7 **Kieli** englanti

Tiivistelmä

Tiivistelmä on lyhyt kuvaus työn keskeisestä sisällöstä: mitä tutkittiin ja miten sekä mitkä olivat tärkeimmät tulokset. Suomenkielisen opinnäytteen tiivistelmä kirjoitetaan suomeksi ja englanniksi ja ruotsinkielisen vastaavasti ruotsiksi ja englanniksi. Suomen- tai ruotsinkielisten opiskelijoiden, joiden opinnäytteen kieli on englanti, tulee kirjoittaa tiivistelmänsä englanniksi ja koulusivistyskielellään. Muiden kuin koulusivistyskieleltään suomen- tai ruotsinkielisten tulee kirjoittaa tiivistelmänsä vain englanniksi. Opiskelija voi halutessaan lisätä opinnäytteeseensä toisen tai kolmannen tiivistelmän omalla äidinkielellään. Tämän opinnäytteen tiivistelmäteksti kirjoitetaan opinnäytteen luettavan osan lomakkeen lisäksi myös pdf-tiedoston metadataan. Kirjoita tähän metadataan kirjoitettavaa teksti. Metadatatekstissa ei saa olla erikoismerkkejä, rivinvaiho- tai kappaleenjakomerkkiä, joten näitä merkkeja ei saa käyttää tässä. Jos tiivistelmäsi ei sisällä erikoimerkkejä eikä kaipaa kappaleenjakoa, voit hyödynttää makroa abstracttext luodessasi lomakkeen tiivistelmää (katso kommentti tässä TeX-tiedostossa alla). Metadatatiivistelmatekstin on muuten oltava sama kuin lomakkeessa oleva teksti.

Avainsanat Vastus, resistanssi, lämpötila

Preface

I want to thank Professor Pirjo Professor and my instructors Dr Alan Advisor and Ms Elsa Expert for their guidance.

I also want to thank my partner for keeping me sane and alive.

Otaniemi, 9 February 2023

Aarni O. Halinen

Contents

Symbols and abbreviations

Symbols

$\uparrow$	electron spin direction up
$\downarrow$	electron spin direction down

Operators

$\nabla \times \mathbf{A}$	curl of vectorin $\mathbf{A}$

Abbreviations

K8s	Kubernetes
STRIDE	an object-oriented analog circuit simulator and design tool

1 Introduction

During the last decade, the modern IT industry has shifted from monolithic software applications towards microservices. In microservice architecture, the application is split into a suite of smaller, independent services that handle some logical part of the business logic [36]. Each service runs its own process and the application data is sent between the components using some lightweight communication mechanisms such as Hypertext Transfer Protocol (HTTP). The architectural approach also increases software agility because each service becomes an independent unit of development, deployment, operations, versioning, and scaling [45]. This modularity is often associated with benefits like faster delivery and improved scalability.

During the same time period, containers have become widely used method for deploying applications. Containers are a lightweight virtualization technique in which the application is bundled with all its dependencies into a single, deployable unit that executes on top of the host machine kernel [17]. As such, a single microservice is quite often build and deployed as a container. In more complex systems with multiple containers, an orchestrator is often used for managing the workloads. Kubernetes is one of the most widely used of these orcehstrator tools.

Similar to how design patterns emerged from the birth of object oriented software systems, the modularity of containers and microservices have enabled the development of distributed system design patterns [18]. The most common of these patterns is the sidecar pattern, in which peripheral tasks like logging and observability are split away as own containers from the main application container. Basically, sidecar pattern is an extension of the modularity of the microservices architecture inside the microservice itself. The benefits of sidecar pattern are similar to microservices, allowing better resource allocation, re-use of components in other services and provide a failure containement boundary, for example. In Kubernetes, the basic unit of deployment is called a Pod, which may include one or more co-scheduled containers. The Pod is analogous to a microservice, while it contains one main container and all its sidecars, bundled into a one deployable unit.

1.1 Problem Statement

While the sidecar pattern makes it easier to add peripheral tasks to applications, it opens up questions about application security. Quite often, developers rely on containers created by third parties for the sidecar tasks. The source code of these sidecar containers, even if it were open, can be hard or even impossible to verify for known vulnerabilities. Furthermore, malicious actors can use supply chain attacks and typosquatting to trick victims into installing malicious sidecars to their clusters. Once malicious attackers gain access to the sidecar, any misconfiugrations or permissive security mechanisms put the whole cluster at risk. Furthermore, Kubernetes is not secure by default; on a fresh installation, most of the included security mechanisms are on permissive settings or outright disabled.

In Kubernetes, there are limited amount of security features available on container-level. Most of the security related policies and capabilities are defined for the Pod,

which essentially means that any capability required by the main application is inherited in the sidecar. Thus, any privileged workload, even in another container in the Pod, risks privilege escalation from the sidecar. In addition, Kubernetes' firewalling solution, Network Policies, are granted for the whole Pod instead of individual containers. Both of the aforementioned issues allow paths for lateral movement and further escalation for the attackers. Thus, any exploitable security issue in a sidecar container makes an optimal launchpad for attack against the whole cluster.

Zero trust architecture and the principle of least privilege are common security paradigms for limiting lateral movement and further escalation in a system if any component within has been compromised. In both of the paradigms the capabilities of an individual component are limited to only those that are required for the component to function. The capabilities, like network access and any container privileges, are explicitly given to the component that requires them, while everything else is denied.

If these paradigms are successfully applied to sidecars, sidecars could only use operations and network access required, while anything else would be blocked. However, since Kubernetes provides limited security on container-level, we need to find some other ways to implement these paradigms inside the Pods. This thesis proposes a solution for restricting the capabilities of sidecar while minimally affecting the main container, thus improving the security by extending the paradigms within the Pod.

1.2 Thesis outline

The following chapter 2 gives background about containers, Kubernetes and explains their common attack vectors. It also discusses Kubernetes networking and container network interface plugins. Chapter 3 proposes ideas for isolating sidecars from main application container. The chapter discusses both container and networking security in the context of Kubernetes Pod. Chapter 6 introduces an implementation based on the findings of the previous chapter. The pros and cons of the solution are discussed in Chapter 7. Finally, Chapter 8 discusses future research and concludes the thesis.

2 Background

2.1 Zero trust architecture

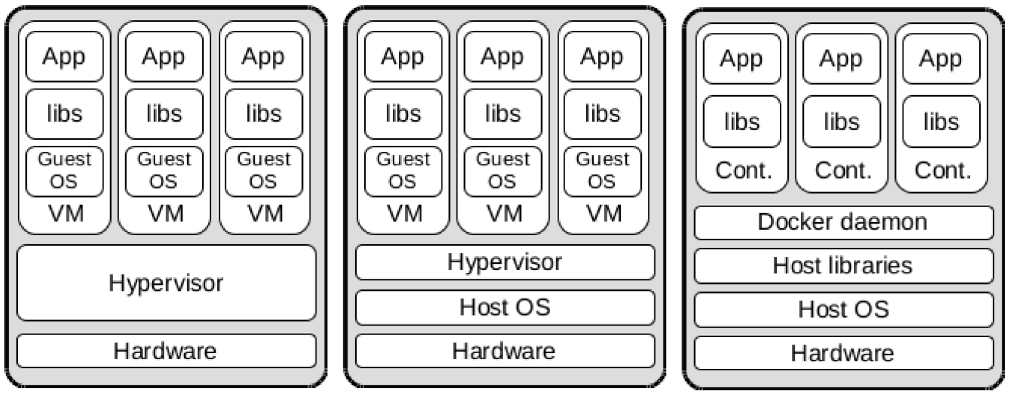
Conventional network security has historically focused on perimeter defence [46]. Subjects like workload resources and users inside the perimeter often assumed to be trusted and implicitly given access inside the network, while any request originating outside the network is subject to more scrutiny. While the systems seems initially secure, the modern IT landscape with cloud-based systems, third party components, remote workers etc. means that the perimeter can be breached via numerous attack vectors. Once any subject inside the perimeter is compromised, the attackers can gain access to all the resources that the subject is authorized to access and move laterally within the perimeter, escalating the attack on other resources.

Zero trust architecture (ZTA) is a security paradigm that focuses on data and resource protection and on the premise that trust must always be explicitly granted and continously evaluated [46, 53]. In contrast to a single perimeter defence, the focus in ZTA is to create a fine-grained access rules around each of the resources, while at the same time enforcing rules that deny other access which is not explicitly allowed. Following the principle of least privilege, the access rules are made as granular as possible so that number of trusted subjects equals to the actual number of subjects that require the access. This achieves a multi-layered security boundaries, where the breach of one component through the most outward perimeter does not compromise the whole system. Instead of having permission to access all resources inside the perimeter, malicious actors could only laterally move to those resources that were required by the compromised component to function. Any other component is still protected by its own perimeter that would require another successful attack to be breached. Thus, the compromised component is of limited usefulness to the attacker instead of serving as a general attack vector against the system.

2.2 Containerization and Docker

Figure 1 illustrates common virtualization models. Whereas traditional virtualization techniques virtualize workloads on top of a hypervisor which shares hardware resources between the virtual machines, containerization is a technique where virtualization happens on a operating system level [49]. Processes executing in containers run on the host machine kernel. However, each container is isolated to its own network, process namespace and so on; two containers on the same host OS do not know that they share resources. Furthermore, containers are similarly isolated from accessing host OS resources.

BSD jails and *chroot* can be considered early forms of containerization technology, so the idea of containers is not new [21]. Recent Linux container solutions rely on two main implementations: Linux Containers (LXC) -based solution that relies on kernel features such as control groups (cgroups) and namespaces, and a custom kernel and Linux distribution called Open Virtuozzo (OpenVZ). Docker [33] is a hugely popular container runtime that is based on LXC and provides an easy-to-use API and tooling



(a) Type1 hypervisor (b) Type2 hypervisor (c) Container

Figure 1: Virtualization models [21]

for creating and managing containers. Docker also provides containerization for other OSes as well. However, in this thesis we focus only on the Linux implementation.

2.2.1 Linux containers

The Linux containers technology implements container isolation and containement using Linux kernel feature called namespaces [47]. Namespaces [30] are a construct that wraps a global system resource in an abstraction which makes it appear to the processes in the namespace that they have their own, isolated, instance of the global resource. There are total of eight namespaces: i) Cgroup which is used for resource management, ii) Inter-process communication (IPC) which isolates POSIX message queues etc., iii) Network which isolates network devices, stack ports etc., iv) Mount for file system isolation, v) Process ID (PID), vi) Time, vii) User for isolating user and group identifiers and viii) UTS which isolates hostnames and NIS domain names. For example, network namespace provides each container their own loopback device and even `iptables` rules. In another example, mount namespace makes sure that container has no visibility nor access to the host's or other container's file system. Compared to other namespaces that concern isolation of kernel data, cgroups focuses on limiting available system resources per namespace [47]. Each namespace can be setup with their own limits on CPU and memory usage and available devices. Using Docker as an example, setting *–cpu*, *–memory* and *–devices* options will limit available resources for the container.

Since all containers and the host machine run on same kernel, any container that manages to breakout from isolation may compromise other containers, the host and the whole kernel. To combat this container breakout, several Linux kernel security mechanisms are adopted to constrain the capabilities of containers [47]. The mechanisms include Discretionary Access Control (DAC) mechanisms like Capability

[29] and Secure computing mode (Seccomp) [31], and Mandatory Access Control (MAC) mechanisms like Security-Enhanced Linux (SELinux) and AppArmor [1]. With Capability, the superuser (i.e. the root user) privilege is divided into distinct units, each of which represent a permission to process some specific kernel resources. The feature turns the binary "root/non-root" security mechanism into fine-grained access control system, which makes it easier to follow the principle of least privilege. For example, processes like web servers that just need to bind on a Internet domain privileged port (numbers below 1024) do not need to run as root; they can just be granted with `CAP_NET_BIND_SERVICE` capability instead [34]. The Seccomp mechanism constrains which system calls a process can invoke. The available system calls are defined for a container through Seccomp profile which is defined as a JSON file. The Docker default Seccomp profile [32] includes over 300 system calls. SELinux is integrated to CentOS/RHEL/Fedora distributions and utilizes a label-based enforcement model, while AppArmor is available in Debian and Ubuntu distros and adopts a path-based enforcement model [47].

2.2.2 Docker

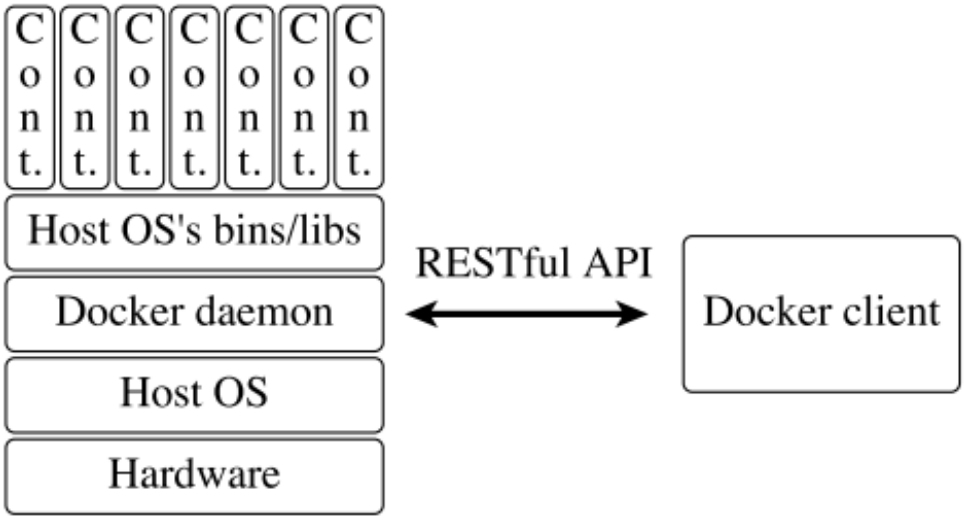


Figure 2: Architecture of Docker engine [17]

Docker is an open-source container technology written in Go and launched in 2013 [3, text]. The platform consists of Docker Engine packaging tool, Docker image registries like the public image repository Docker Hub and Docker desktop application [2]. In general, the engine architecture is similar to container-based virtualization, as visible in figure 2 [17]. The containers run on top of Docker daemon which manages and executes all the containers. The daemon is exposed to Docker clients via RESTful HTTP API. The Docker client is a command line tool which provides user interface

for commanding the daemon and thus containers. By exposing the API outside the host machine, the architecture enables remote control of daemon with the client. For security reasons, remote communication should be secured with TLS.

Docker image is a read-only template with instructions for creating a Docker container [2]. The images are often based on another image, such as OS images `ubuntu` and `alpine`, with some additional customizations like installation of web server binaries. The customizations are added to the image as series of data layers so that each new command creates a new layer. This process makes the distribution of image more efficient since only the changes between layers need to be distributed [17]. The layering is achieved with special filesystem inspired by UnionFS which allows files and directories in different file systems to be combined into a single consistent file system.

Docker users can share their custom images publicly or privately in Docker Hub, or even host their own image registry platform. Most cloud providers also offer container registry services so even propietary software can be published in a private registry and used by other cloud services, like Kubernetes clusters. Whenever the image is not found locally, the client automatically tries to search and pull image from connected registries.

2.3 Kubernetes

Kubernetes (K8s) [8] is an open-source container orchestrator, i.e. a system for automating deployment, scaling and management of containerized applications. It allows creation of a cluster which consists of a set of servers, called Nodes, on which application containers are scheduled by the system. The automation provides resilience and efficient resource utilization for workloads in the cluster: if a container or node dies, the system tries to restart and re-schedule containers so that the desired cluster state is maintained. K8s is hosted by the Cloud Nativce Computing Foundation (CNCF), but it's origins are at Google where it was created as an open-source option for Google's propietary Borg and Omega orchestrators [19]. K8s was open-sourced in 2014.

2.3.1 Kubernetes objects

Pods are the basic atomic scheduling unit in K8s. Pods consists of one or more tightly-coupled containers with shared storage volume and networking [12]. Containers in a pod are always co-located and co-scheduled and run in a shared context, i.e. a set of Linux namespaces. Network, UTS and IPC namespaces are shared by default, and process namespace can be shared with `v1.PodSpec.shareProcessNamespace`. The common network namespace means that containers in a pod can communicate with each other via localhost, have common IP address and cannot re-use same port numbers. In addition to normal application container, Pods can include special `initContainers` that are only run on Pod startup. These pods are used for modifying Pod context before the actual workload starts. Multiple `initContainers` are run sequentially and a failing container blocks the execution of the following initialization

and normal workloads. All Pods accross the cluster share same subnet and can access each other via IP address. However, connecting to a Pod with IP address is sub-optimal since Pods are ephermal and restarting a dead pod may receive a new IP address. Furthermore, horizontally scaled Pods with multiple replicas have as many IP addresses, thus making load balancing difficult. Kubernetes concept called Services solves these issues.

Instead of creating Pods directly, **Deployment** workload resources are used for creating Pods in a cluster, even with singleton Pods [12]. With Deployments, user describes the desired state in a declarative manner. The Kubernetes control loop then creates **ReplicaSet** based on the Deployment resource, which in turn guarantees the availibility of desired amount of Pods [7]. **DaemonSet** on the other hand is a workload resource that ensures all or some Nodes run a copy of a Pod. Typical usecases for daemons are running Node monitoring and logging, and network plugins which was discuss in depth in section 2.4.1.

Services are an object for exposing groups of Pods over an network [14]. The object defines a set of enpoints, i.e. the targeted pods, along with a policy about how to make the pods accessible. The targeted pods are determined with a `selector` field in the object specification. Meanwhile, the `type` field determines how the Service is exposed. There are four different `ServiceTypes` levels: i) the default `ClusterIP` which exposes Service inside the cluster with its own IP address, ii) `NodePort` which exposes service in each Node's IP address on static port (by default within a range of 30000-32767), iii) `LoadBalancer` which exposes the Service externally using cloud provider's load balancer and iv) `ExternalName` which is used to map Service to DNS name instead of a group of Pods. The field is designed as a nested functionality; each `ServiceType` level adds up to the previous one. Ingress object can also be used for exposing Services to outside of cluster. The Ingress object requires installation of an Ingress Controller to the cluster. Cloud providers often have their own controllers and all the examples in this thesis are executed on a local cluster where no external access is needed. Thus, the controllers are left as an exercise for the reader.

Namespaces provide isolation for cluster objects and allow grouping of objects under a single name. New K8s cluster starts with four namespaces: `default`, `kube-node-lease`, `kube-public` and `kube-system`. Namespaced objects like Deployments, Services and Pods are always deployed under a namespace which is `default` if not explicitly defined. `kube-system` is the namespace for all objects created by the K8s system which is discussed more in depth in the next section 2.3.2. Namespaces also provide a scope for naming; names of resources need to be unique within a namespace, but not across namespaces. Namespaces are also used to enforce resource quotas, access control, and isolation for cluster users, for example in multi-tenancy setups. Pod Security Standards [11], which are used by Pod Security admission controller, are also defined at namespace level. Admission controllers are discussed in section 2.3.3.

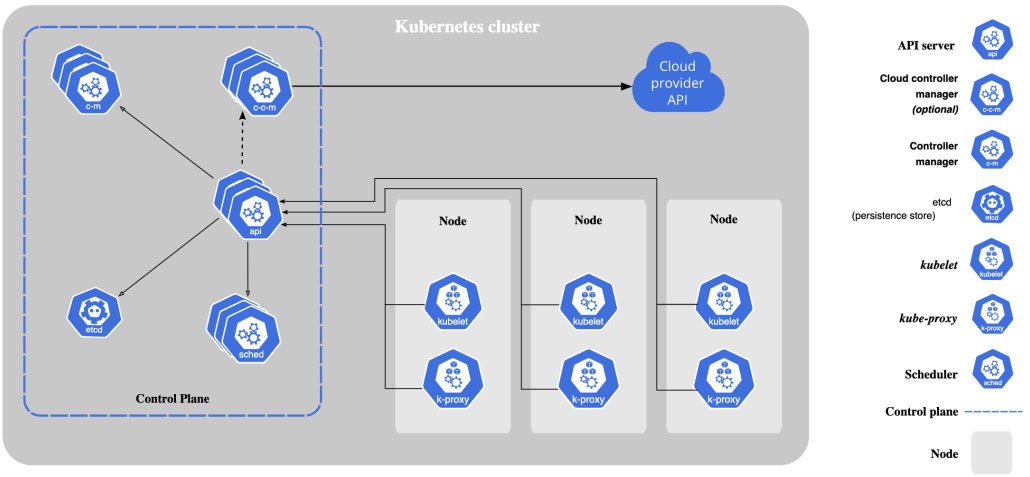


Figure 3: Kubernetes cluster architecture [9]

2.3.2 Kubernetes components

The figure 3 describes Kubernetes cluster with control plane and three worker nodes. The control plane consists of components that control, monitor and store the state of the cluster; essentially these are the components that are needed for complete and working Kubernetes cluster [9]. The control plane components can be run on any worker node. However, clusters often have specialized master node for control plane components, which does not run any other containers. For fault-tolerance and high availability, control plane components should run on multiple Nodes in production environments. The control plane consists of these main components:

API server is a frontend component of the control plane. It is a stateless HTTP server which validates and authenticates commands given to the cluster. For valid commands, the server then forwards these to other control plane components. For example the `kubectl` CLI tool sends commands to the API server with HTTP. The main implementation of the server is `kube-apiserver`. The server can be horizontally scaled by running several instances on multiple Nodes and load balancing traffic between the instances.

Etcd [4] is a strongly consistent, distributed key-value store. It is the stateful component of the control plane: all of the cluster data is stored in etcd. Thus, the stability of the component is critical for the whole cluster. To tolerate failures, etcd implements a leader-based architecture. Multiple etcd clients automaitcally elect a leader instance as the source of truth. Other instances periodically update their state from the leader instance, so that the state stays eventually conistent across all the instances. On leader failure, the other instances automatically elect new leader to keep the system functioning.

Scheduler watches for newly created Pods that have no assigned worker node, and selects one of the active Nodes for them to run on [9]. The scheduling takes into account resource availibility on Nodes, Pod resource requirements, object specification affinity rules and hardware, software and policy constraints, among others.

Controller manager is a control plane component that runs all the controller loop processes [9]. Controller loops, like the Deployment controller, continously watch the current and desired cluster state. When the states differ, they send commands via the API server so that the cluster moves towards the desired state. All the built-in controllers are compiled into a single binary, even though the controllers are logically different processes.

Each Node also has components that are essential for Kubernetes to work properly. **Kubelet** is an agent that makes sure containers are running in a Pod [9]. It receives a set of pod specification from the API server and ensures that containers are running on the Node, follow the pod specifications and are healthy. Any container which is not created by Kubernetes is not managed by `kubelet`. **Kube-proxy** maintains network rules on Nodes. Part of the Serivce objects' networking is implemented by **kube-proxy**; the proxy writes `iptables` rules that route the traffic [25].

2.3.3 Admission controllers

Admission controllers are a feature of Kubernetes API server, made for validating and modifying requests made to the server [5]. The controllers execute prior to persistence of the object, but after the request is authenticated and authorized by the server. Several important features of Kubernetes are implemented with admission controllers, and these should be enabled in a properly configured API server. In addition to the built-in controllers, Kubernetes provides `MutatingAdmissionWebhook` and `ValidatingAdmissionWebhook` controllers for building own admission logic.

Admission controllers can be validating, mutating, or both [5]. Mutating controllers may modify related objects to the requests they admit, while validating controllers either approve or reject the request. The control process executes mutating controllers first so that no mutations happen after the validation. If a controller in either phase rejects the request, the request is not processed further and error is returned to the end-user.

The important Admission controller for the scope of this thesis is the `PodSecurity` controller. The admission controller validates Pods before they are admitted, making sure that the requested Pod security context and other restrictions are permitted in the namespace that the Pod is assigned to [5]. The controller is enabled by default, and can be taken into use just by configuring Pod Security Admission labels for Namespace objects.

The labels use `pod-security.kubernetes.io/<MODE>: <LEVEL>` format, where MODE defines the action to be taken when the security level is violated and the LEVEL is a pre-defined Pod Security Standard level. The three available levels are `privileged`, `baseline` and `restricted` [11].

The actions available are i) `enforce`, which will reject the pod on violation, ii) `audit`, which triggers an event about the violation in the audit log, and iii) `warn`, which triggers user-facing warning about the violation [10]. A namespace can configure any or all three of the available modes, and even set a different level for the modes. For example, it is possible to warn the user about security policy violation without blocking the request by setting the `warn` mode more restrictive than the `enforce`.

2.3.4 Sidecar pattern

As mentioned before, Pods are the basic scheduling abstraction in Kubernetes and they support management and co-scheduling of multiple containers as an atomic unit. This co-scheduling and management of multiple symbiotic containers as a single unit enables multi-container application design patterns to emerge [18]. Sidecar pattern is the most common of these design patterns. As an example of this pattern, the main application container can be a simple web server pairted with a container that collects server's log file and streams them to a centralized log management system. Another example of this pattern is Istio service mesh [28] and its Envoy proxy sidecar which routes all traffic thorugh the Istio control plane for management, observability and security reasons.

In the pattern, peripheral tasks such as logging, configuration and observability are isolated from the main application into helper containers. These containers, sidecars, are tightly-coupled to the parent application container and should share the lifecycle of the parent. Even though the functionality of the sidecars could be build into the main container, there are benefits for using separate containers [18]. The isolation allows tweaking of containers' cgroups so that CPU cycles can be prioritized for the main container. The isolation also provides failure containement boundary between the main and sidecar processes. Since container is also the unit of deployment, the sidecar containers could be developed, tested and deployed independently from one another. Sidecar containers can also be developed with different tools and dependencies, and in a way that they are re-usable with other application containers. However, this multiplies the amount of moving parts of the overall system, which increases the size of test matrix considering all of the container version combinations that might be seen in the production environment.

2.4 Kubernetes network model

Integral part of Kubernetes cluster is how nodes and resources are networked together. Specifically, the networking model needs to address four different type of networking problems: i) intra-Pod (ie. container-to-container within same Pod) communication, ii) inter-Pod communication between Pods, iii) Service-to-Pod communication and iv) communication from external sources to Services [6]. The model also requires that each Pod is IP addressable and can communicate with other Pods without network address translation (NAT), even when Pods are scheduled on different hosts [52]. All agents on a host should also be able to communicate with Pods on the same host. The implementation of this model is not part of Kubernetes, but is handed to special plugins that implement Container Network Interface (CNI) specification.

2.4.1 Container Network Interface

The Container Network Interface (CNI) [15] is a networking specification, which has become de facto industry standard for container networking. It is backed by CNCF [52]. CNI was first developed for the container runtime `rkt`, but it is supported

by all container runtimes and there is a large number of implementations to choose from [43]. Most of the container orchestrators have adopted the specification as their networking solution. The biggest outlier is Docker Swarm, which instead implements `libnetwork` [35].

The CNI specification has five distinct definitions: i) a format for network configuration, ii) a execution protocol between the container runtimes and the plugin binary, iii) a procedure for the runtime to interpret the configuration and execute the plugins, iv) a procedure for deletegating functionality between the plugins and v) data types for plugins to return their results to the runtime [15]. The network configuration is defined as a JSON file and it includes a list of plugins and their configuration. The container runtime interprets the configuration file at plugin execution time and transforms it into a form to be passed to the plugins. The execution protocol defines a set of operations (ADD, DEL, CHECK, VERSION) for adding and removing containers from the network. The operation command, similarly to other protocol parameters, are passed to the plugins via OS environment variables. The configuration file is supplied to the plugin via stdin. On successful execution, the plugin returns the result via stdout with a return code of 0. On errors, the plugin returns a specific JSON structure error message to stderr and a non-zero return code. When the runtime mutates a container network, it results in a series of ADD, DELETE or CHECK executions. These are then executed in same order as defined in the `plugins` list, or reversed order for DELETE executions. Each plugin then returns either `Success` or `Error` JSON object. The execution of a series of operations ends when it encounters the first Error response, or when all the operations have been performed.

The CNI plugin must provide at least connectivity and reachability for the containers [42]. For connectivity, each Pod must have a NIC for communication outside its networking namespace. The NIC must have IP address reachable from the host Node, so that cluster processes like Kubelet health and readiness checks can reach the Pod. Reachability means that all Pods can be reached from other Nodes directly without NAT. Thus, each Pod receives an unique IP address from the `PodCIDR` range configured on the Node by the Kubelet bootstrapping phase. The end-to-end reachability between different Node `PodCIDR`s is established by encapsulating in the overlay network (for example with VXLAN) or orchestrating on the underlay network, e.g. with Border Gateway Protocol (BGP).

Since Kubernetes does not provide networking between the Pods, it has no capabilities to enforce network isolation between workloads. Thus, another key feature for CNI plugins is enforcing network traffic rules. Kubernetes provides a common object called `NetworkPolicy` for CNI plugins to consume. The NetworkPolicy specification consists of a `podSelector` that specifies pods that are subject to the policy and `policyTypes` to specify Ingress and Egress rules for the traffic [16] to the target Pod. Each rule includes `to` or `from` field for selecting Pod, Namespace or IP address block in CIDR notation on the other side of the connection, and `ports` field for explicitly specifying which ports and protocols are part of the rule. The policies are additive; when multiple rules are defined for a Pod, the traffic is restricted to what is allowed by the union of the policies. Many CNI plugins also introduce Custom Resource Definitions for their own, more granular, network policy rules.

While all CNI plugins meet the requirements listed above, they may differ in architecture significantly. The plugins can be classified based on which OSI model network layers they operate on, which Linux kernel features they use for packet filtering and which encapsulation and routing model they support for inter-host and intra-host communication between Pods. In this thesis, we focus on three different CNI plugins: Calico, Cilium and Multus.

2.4.2 Calico

Calico [55] is an open-source CNI plugin with modular architecture that supports wide range of deployment options. Each Pod created to the Calico network receives one end of a virtual ethernet device link as its default `eth0` network interface, while other end is left dangling on the host Node [40]. The Pod end of the link receives IP address from Pod CIDR, but the Node end does not. Instead, a `proxy_arp` flag is set on the on the host side of the interface while containers have a route to link-local address `169.254.1.1`, thus making the host behave like a gateway router. For routing packets between Nodes, Calico creates a VXLAN overlay network. Optionally, Calico supports IP-in-IP overlay or non-overlay network with BGP protocol.

On each Node, a `calico-node` daemon setups CNI plugin, IPAM and possible eBPF programs. The daemon subscribes to Kubernetes API for Pod events and manages both container and host networking namespaces. Calico also deploys a single-container `calico-kube-controllers` Pod into the Kubernetes control plane. The container executes a binary that consists of controller loops for Namespace, NetworkPolicy, Node, Pod and ServiceAccount Kubernetes objects. The Calico project also introduces own CLI tool, called `calicoctl` [57], for managing Calico's custom resources. The tool provides extra validation for the resources which is not possible with `kubectl`.

Calico supports Kubernetes NetworkPolicies as well as its own namespaced `projectcalico.org/v3.NetworkPolicy` Custom Resource Definition (CRD). Both of the policies work on OSI layers L3 (identity, e.g. IP address) and L4 (ports). Compared to the built-in policy, the Calico policy includes features such as policy ordering, log action in rules, more flexible matching criteria (e.g., mathcing on ServiceAccounts) [56]. The policy can also match on other Calico CRDs such as **HostEndpoints** and **NetworkSets**, which allows implementing rules on host interfaces and non-Kubernetes resources. If Calico is installed along Istio service mesh, the Calico Network Policy can enforce L7 (e.g. HTTP methods and URL paths) policies on the Envoy proxy. For policies that are not tied to a Kubernetes namespace, Calico provides a `GlobalNetworkPolicy` CRD.

2.4.3 Cilium

Cilium [22] is one of the most advanced and powerful CNI plugins for Kubernetes. Similarly to Calico, it creates virtual ethernet device for each Pod and sets one side of the link into Pod's network namespace [41] as the default interface. Cilium then attaches extended Berkeley Packet Filter (eBPF) programs to ingress traffic control (`tc`) hooks of these virtual ethernet devices for intercepting all incoming packets

from the Pod. The packets are intercepted and processed before the network stack and thus `iptables`, reducing latency 20%-30% and even doubling the throughput of packets in some scenarios [16]. The network between Pods running on different hosts is handled by default with VXLAN overlay, but there is support for Geneve interfaces and native-routing with BGP protocol as well [22].

The Cilium system consists of an agent (`cilium-agent`) daemon running on each Node, one or more operator (`cilium-operator`) Pods and a CLI client (`cilium`) [23]. The agent daemons subscribe to events from Kubernetes API and manage containers' networking and eBPF programs. The CLI tool, which is installed on each agent, interacts with the REST API of the agent and allows inspecting the state and status of the local agent. The tool should not be confused with Cilium management CLI tool, also incidentally named `cilium`, which is typically installed remote from the cluster. The operator is responsible for all management operations which should be handled once for the entire cluster, rather than once for each Node. This includes for example registering of CRDs.

While default Kubernetes Network Policy provides security on OSI layers L3 and L4, Cilium provides CRDs that also support for L7 policies [24]. If L7 policies exist, the traffic is directed to Envoy instance bundled into the agent Pod which filters the traffic. Unlike on layers 3 and 4, policy violation does not result in dropped packet but an application protocol specific denied message. For example, HTTP traffic is denied with `HTTP 403 Forbidden` and DNS requests with `DNS REFUSED`. Cilium provides `CiliumNetworkPolicy` CRD that supports all L3, L4 and L7 policies. Cilium also provides `CiliumClusterwideNetworkPolicy` custom resource which is used to apply network rules to every namespace in the cluster or even to nodes when using `nodeSelector`.

As even more advanced features, Cilium also includes natively `kube-proxy` replacement, encryption for Cilium-managed traffic and Service Mesh, among others. By default, `kube-proxy` uses `iptables` to route the Service traffic [25]. With `kubeProxyReplacement` installation option, Cilium implements Service load-balancing as XDP and TC programs on Node network stack. For encryption, Cilium supports both IPsec and WireGuard implementations [26]. The Service mesh performs variety of features directly in eBPF, thus functioning without sidecar containers or proxying requests through the agent Pod's Envoy [37]. Since all features are not available as eBPF programs or on all kernel versions, Cilium automatically probes the underlying kernel and automatically reverts to Envoy proxy when needed. For capabilities beyong the built-in mesh, Cilium also provides an integration with Istio.

2.4.4 Multus

Traditionally CNI plugins only provide a single network interface for a Pod, apart from loopback device. Multus [38] is a CNI plugin that allows attaching multiple network interfaces for a Pod. It does not provide any connectivity or reachability for the containers like other plugins. Instead, it is installed as the first plugin in the CNI plugin chain. When executed, the plugin delegates interface creation to other installed plugins. Since Multus does not provide any networking and thus does not

independently, it is often called "meta plugin" to distinguish it from common CNI plugins like the previous Calico and Cilium.

Multus system includes a binary, a CNI configuration file and a namespaced `NetworkAttachmentDefinition` CRD that is used to define network interfaces used in Pods. The binary and the configuration file are often installed to cluster Nodes via a DaemonSet. The daemon consists of an *initContainer* that copies the binary into the `/opt/cni/bin` directory, and a daemon container that setups the configuration file and optionally spawns a HTTP server for additional features such as metrics [38]. The configuration file satisfies the CNI specification with few extra attributes of which the combination of `clusterNetwork` and `defaultNetworks` or `delegates` are imperative for the CNI plugin to function [39]. The `clusterNetwork` specifies the main network of the cluster, which implements the `eth0` interface and Pod IP address. The `defaultNetworks` is an optional array of networks that should be added for any Pods by default. The values can be names of the `NetworkAttachmentDefinition` objects or paths to CNI plugin's JSON configuration files. Optionally, the `delegates` attribute can be used; it supports similar format of values. In this scenario, the first element of the array functions as `clusterNetwork` and the rest are infered as `defaultNetworks`.

Attaching additional interfaces to workloads is most often configured by adding a special annotations field `k8s.v1.cni.cncf.io/networks` to workload resource definitions. In the simplest configuration, the field takes a comma-separated list of `NetworkAttachmentDefinition` names as input. The network interface identifiers can be modified by giving the attachment input in *name@interface-identifier* format. Otherwise, Multus names the interfaces *net0*, *net1* and so on. If extra configuration for the networks is needed, the annotation also supports a JSON array format.

2.4.5 Extended Berkeley Packet Filter

Berkeley Packet Filter (BPF, or nowadays often cBPF) was originally developed in early 1990s as a high-performance tool for user-space packet captures [48]. BPF works by deploying the filtering part of the application, `packet filter`, in the kernel-space as an agent. The `packet filter` is provided with a program (often denoted as BPF program) consisting of BPF instructions, which works as a set of rules for selecting which packets are of interest in the user-space application and should be copied from kernel-space to user-space. The instuctions are executed in a register-based pseudo machine. Since network monitors are often interested only in subset of network traffic, this limits the number of expensive copy operations across the kernel/user-space protection boundary only to packets that are of interest in the user-space application. A notable usecase for BPF is *libpcap* library, which is used by network monitoring tool called `tcpdump`.

Later in the 2010s the Linux community realized that BPF and it's ability to instrument the kernel could benefit other areas than packet filtering as well [58]. This reworked version of BPF was first merged in to Linux kernel in 2014 and is publicly called extended Berkeley Packet Filter (eBPF) to distinguish it from the original cBPF. The kernel development community continues to call the newer version BPF, but

instead of the original acronym consider it a name of a technology. Similarly to the kernel community, the term BPF always refers to the eBPF in this thesis.

The eBPF programs are compiled to bytecode and loaded to kernel with `bpf()` system call [50]. Most often programs are written in restricted C and compiled with LLVM Clang compiler to bytecode. It also possible to use eBPF assembly instructions and `bpf_asm` utility for converting instructions to bytecode. eBPF programs follow a event-driven architecture: a loaded eBPF program is hooked to a particular type of event and each occurence of the event triggers the program execution.

For networking purposes, there are two eBPF hooks available for intercepting and mangling, forwarding or dropping network packets: eXpress Data Path (XDP) and Traffic Control (TC) [50]. In Cloudflares DDoS testing benchmark [20], XDP program was capable to drop 10 million and TC program 2 million packets per second, while common `iptables` INPUT rule was able to drop less than one million packets per second.

XDP programs are attached to a network interface controller (NIC) and can handle only incoming packets [44]. The programs are called directly by the NIC driver if it has XDP support, thus executing before packets enters the network stack. This skips expensive packet parsing and memory allocation operations, and allow XDP programs to run at very high throughput. Thus, even the main networking buffer *skbuff* is not populated. Some SmartNICs even support offloading the program to the NIC's own processor from host CPU, improving host machine performance even further [27]. If the driver does not support XDP, generic XDP is used and the programs run after the packet has been parsed by the network stack.

XDP programs can read and modify contents of the packets [58]. Since the packets are not parsed the network stack, the programs have to work with raw packets and implement own parsing functionality. The program's return value determines how the packet should be processed further. With `XDP_DROP` and `XDP_PASS` return values, the packet can be dropped or passed further to the networking stack respectively. The packet can also be bounced back to the same NIC it arrived on with `XDP_TX`, usually after modifying the packet contents. `XDP_REDIRECT` is used for redirecting the packet to a different NIC, CPU or even to another socket.

TC programs are executed when both incoming and outgoing packets reach kernel traffic control function within the Linux network stack [58]. The ingress hook executes after the packet is parsed to *skbuff* but before most of the network stack. On egress the stack is traversed in reverse, thus the hook executes after most of the network stack. TC programs can read and write directly to packet in memory. Similarly to XDP programs, the return value of the program determines further processing of the packet. The packet can be passed furhter in the stack with `TC_ACT_OK`, dropped with `TC_ACT_SHOT`, or the modified packet can be redirected back to the start of the classification with `TC_ACT_RECLASSIFY`, among others.

2.5 Example attack scenarios

- Privileged container
- CAP_SYS_ADMIN, mounting /proc and chroot
- CAP_SYS_PTRACE, shellcode injection to running program, nc 172.17.0.1 on port running shell
- Mounted docker socket, creating privileged containers

All of the example attack scenarios start by attacker getting shell access to a container running in a Pod, usually through a remote code exectuion (RCE) flaw on the container application and then executing reverse shell inside the Pod. The scope of the examples is not in the initial attack, but in the Pod template misconfiugrations that then provide some path for the attacker to escalate the attack and even take over the whole cluster in the end.

TODO: add file

The first attack scenario includes a privileged container, as described in file 2.5. Privileged containers have all the capabilities of the host machine, so practically they can be perform almost any action available on the host. This includes but is not restricted to for example "pipeing" (TODO: better word for undock.sh and other scripts) commands as root to the Node's shell and mounting the host's disks. If combined with `hostPID: true`, the attacker can see all the processes on the host, and use `nsenter` to execute commands in the other processes' namespaces.

TODO: add file

The second example file 2.5 does not have similar privileges for executing commands as the first, but has unlimited access for mounting the whole host's filesystem, with both read and write access. Thus, the attacker can try to find any credentials stored on the host machine and use these to escalate the attack. Important credentials include `kubeconfig` files which store access token to K8s API server, ServiceAccount tokens that may have been mounted on any Pod on the host, SSH keys and hashed user passwords in `/etc/shadows`.

TODO: some extra issues (finding .kubeconfig files, running on control-plane -> etcd and secrets within, cloud metadata)

3 Solution requirements

This chapter analyzes threats in K8s cluster from the perspective of sidecars and defines security requirements for solution introduced in following chapters. First, threat modeling is used to identify possible attack vectors and readily available mitigations. Then, the model is used to identify all issues that should be solved with the solution. Finally, an example development environment is introduced for testing out the solution.

3.1 Threat modeling

A security threat is any possible event in a system that could lead to a potential loss of confidentiality and integrity of an asset in the system. Threat modeling is a structured approach to identify and prioritize the potential threats to a system. This includes profiles of potential attackers and their goals and methods, as well as potential mitigations [54]. The idea is to build a catalog of all the potential threats, prioritize the issues and try to find mitigations so that the attackers cannot use the vulnerabilities against the system.

For generic K8s clusters there exists a number of extensive threat models, for example the *Threat Matrix for Kubernetes* by Microsoft [51]. The scope of this thesis is in sidecar security, so we will only focus on the possible threats that can be leveraged after attacker has gained access to a sidecar container inside the cluster. The *Threat Matrix for Kubernetes* is used to identify the threats in this scenario, while the results are listed in the table 1.

Table 1: K8s sidecar threat model

Threat	Description	Mitigation
Privileged containers	If containers are given privileges, malicious actor can breakout from the container and escalate the attack on cluster.	Use `restricted` Pod Security Admission to enforce security rules. PSAs are defined for namespaces, so isolate privileged containers into own namespaces to follow principle of least privilege.
Writing to host file system	TODO.	Disallow unnecessary mounts. Use `readOnlyRootFilesystem: true` whenever possible.
Permissive RBAC on service accounts	Containers in a Pod share service accounts[1]. By default, Pods automatically mount the service account to all containers.	Disable automatic mounting of service accounts[2] and explicitly mount them to containers when neeeded. Adhere to the principle of least privilege.
No resource limits for containers	If not set, single container can hog system resources, causing denial of service (DoS).	Set resource limits to containers[3].
Pod has network access to control plane	By default, all traffic inside cluster is allowed.	Use NetworkPolicies as a firewall. Deny by default, and explicitly allow traffic only if needed.
Sidecar can use main container's NetworkPolicy to access `kube-system` or other Pods	NetworkPolicies affect the Pod network NIC, which is shared by the sidecars.	**No built-in solution available!**
Sidecar has network access to cloud resources	Cloud providers may attach cloud identities to cluster Nodes, so that Pods can access resources outside the cluster.	**Same issue as above!** Block mounting volumes with access to cloud credentials.
Sidecar has network access to the main container	Main container is accessible via loopback device, which cannot be protected with NetworkPolicy.	**No built-in solution available!**
Sidecar is used to sniff main application's traffic	TODO	mTLS? Requires service mesh.
Unencrypted secrets	TODO	Encrypt at rest

3.2 Security requirements

The threat modeling identified two main categories of issues: permissive workload configurations and networking related issues. Essentially, all the threats are caused by sidecars not respecting principle of least privilege; the sidecar inherits execution and networking privileges from the main application container. Based on the model, the solution should provide answer to these main questions:

1. How to ensure and enforce that no workloads that conflict with principle of least privilege are deployed to the cluster?

2. How to enforce Zero Trust network, that allows traffic filtering on container-level and limits communication on both loopback and Pod network interface device?

As for the first question, existing mitigations were already found while threat modeling. A solution where all the mitigations are applied is introduced in chapter 4. For the second question on building the Zero Trust network, the Pod must be firewalled for both inter-Pod communications on the Pod network NIC, and intra-Pod communications on the loopback NIC. However, K8s does not provide any built-in solution for creating these firewalls.

The root cause for the issues is that the lowest level of networking abstraction in K8s is the Pod, and that by definition, all containers in a Pod share network namespace. Thus, all traffic outside the Pod, from any of the containers, goes through the same `eth0` NIC and shares common source IP address. Since NetworkPolicies function on L3/L4, they cannot distinguish whether traffic originates from the sidecar or from the main container. Even though ingress traffic can be identified trivially with destination port number, the source port for egress traffic depends on the TCP implementation. As a result, NetworkPolicies cannot be used to manage egress traffic from sidecar independently of the main container. For the intra-Pod communication, the traffic goes through the loopback device which is not managed by CNI plugins. This means that NetworkPolicies have no effect on the loopback device. Furthermore, as all intra-Pod traffic has source and destination IP addresses of `127.0.0.1`, the packets have no clear identifiers that could be of used for traffic filtering. Few possible solutions for building the Zero Trust network inside the Pod is given in chapter 5.

3.3 Environment

The solution is tested and developed on a `Minikube` cluster running on local machine and using Docker as the driver for Nodes. The cluster consists of two Nodes, the purpose of which is to deploy control plane components separately from worker ones. Furthermore, the setup also allows testing of network between components hosted on different Nodes. The complete setup is hosted on Github (TODO: add link). Both Calico and Cilium are used as CNI plugins, since the selection of CNI plugin has minor implications on the actual networking solution. The implications are discussed in detail in chapter 5.

A simple Node.js webserver with StatsD sidecar is used as the application workload. The source code can be found in appendix B. The webserver serves "Hello world" on port `8888` and writes response times to the StatsD sidecar via loopback device on port `8125`, which is the default for StatsD.

For penetration testing purposes, another sidecar with common networking and K8s command line tools is introduced. When deployed instead of the StatsD client, this sidecar can be used to simulate situations where the attacker has managed to get access to the shell inside the sidecar. The Dockerfile for the image can be found in appendix A.

4 Hardening Pod security

This chapter provides a solution for hardening Pods against privilege escalation attacks. The solution enforces that deployed resources follow K8s best security practices. Most of the practices are enforced by the built-in Pod Security Admission controller. However, this chapter also introduces extra security measures that fix few oversights regarding sidecar containers in the controller.

4.1 Restricted Pod Security Standard

Since version 1.25, K8s has shipped with Pod Security Admission controller as a stable feature. The controller, as discussed in chapther 2.3.3, provides three different Pod Security Standards that can be used to warn and enforce against insecure Pod configurations. The `restricted` Pod Security Standard is the strictest of the standards and aims on the best Pod hardening practices [11], so it will be the most optimal for our solution. The table 2 lists all fields affected by the standard.

Table 2: Pod fields enforced by `restricted` Security Standard

Field name	Usage	Allowed values
hostPID, hostIPC, hostNetwork	Controls whether container uses host's PID, IPC and network namespace.	false
privileged	Controls whether Pod can run privileged containers.	false
capabilities.add	Defines Linux capabilities for the container.	NET_BIND_SERVICE
capabilities.drop	Defines Linux capabilities for the container.	ALL
volumes[*]	All volume types are not allowed. For example, hostPath, that maps host directories, are not allowed.	`volumes[*].configMap`, `volumes[*].csi`, `volumes[*].downwardAPI`, `volumes[*].emptyDir`, `volumes[*].ephemeral`, `volumes[*].persistentVolumeClaim`, `volumes[*].projected`, `volumes[*].secret`
hostPort	Expose container via host's network port.	undefined
container.apparmor.security.beta.kubernetes.io/* annotation	Sets the AppArmor profile used by containers. On supported hosts, the runtime/default AppArmor profile is applied by default.	runtime/default, localhost/*
seLinuxOptions	Sets the SELinux context of the container.	Set if supported by environment.
procMount	The default /proc masks are set up to reduce attack surface, and should be required.	Default
seccompProfile.type	Sets the seccomp profile used to sandbox containers.	RuntimeDefault or Localhost
sysctls[*].name	Sysctls can disable security mechanisms or affect all containers on a host, and should be disallowed except for an allowed "safe" subset.	kernel.shm_rmid_forced, net.ipv4.ip_local_port_range, net.ipv4.ip_unprivileged_port_start, net.ipv4.tcp_syncookies, net.ipv4.ping_group_range
allowPrivilegeEscalation	Restricts escalation to root privileges.	false
runAsNonRoot	Controls whether container can run as root user.	true
runAsUser, runAsGroup	Controls the user and	Set both to non-zero

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: foo
  labels:
    name: foo
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/enforce-version: latest
    pod-security.kubernetes.io/audit-version: latest
    pod-security.kubernetes.io/warn-version: latest
```

Listing 1: Namespace resource with `restricted` Security Standard

The Security Standard can be enforced trivially by adding `pod-security.kubernetes.io/enforce: restricted` as label on a K8s namespace resource, as shown in example 1.

4.2 Enforcing other best practices

The `restricted` Security Standard hardens the Pod against most of the identified security threats. However, it still does not enforce specific resource limits and allows automatic mounting of Service Accounts for the Pod containers.

Adding resource limits to containers is straight-forward: just add values to both `resources.limits.cpu` and `resources.limits.memory` fields for all of the containers, similarly to example 3. The CPU usage is measured in CPU units and can also be expressed in millicpus, ie. both "1000m" and integer value of 1 are equivalent to 1 physical or virtual core [13]. For memory, the base unit is bytes, but it also supports quantity suffixes like "M", "Mi" and "Gi" for megabytes, mebibytes and gigibytes, respectively. The resource limits are registered to container's cgroup by the Kubelet. The limits are hard, which means that if a container exceeds its CPU limit, the execution is blocked until more CPU capacity is available. Exceeding the memory limit causes termination with a out-of-memory (OOM) error.

4.2.1 Manual service account mounting

By default, the Service Account tokens are mounted into `/var/run/secrets/kubernetes.io/servicea` directory in every container. This feature can be disabled by setting `automountServiceAccountToken: false`, but then any container that actually uses the Service Account must receive the token some other way. Since volume mounts are defined per container, and service account tokens can be created manually with Secrets, the issue can be circumvented by manually mounting the token to containers that use it.

The example 2 shows how to create a Service Account token with permission to

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: foo
  name: foo-service-account-role
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: foo-service-account
  namespace: foo
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: foo-service-account-rolebinding
  namespace: secure-ns
subjects:
- kind: ServiceAccount
  name: foo-service-account
  apiGroup: ""
roleRef:
  kind: Role
  name: foo-service-account-role
  apiGroup: ""
---
apiVersion: v1
kind: Secret
metadata:
  name: foo-service-account-token
  namespace: foo
  annotations:
    kubernetes.io/service-account.name: foo-service-account
type: kubernetes.io/service-account-token
```

Listing 2: ServiceAccount with permissions for fetching Pod resources

read the status of Pods in the cluster. While Role, RoleBinding and ServiceAccount resources are used to create and define RBAC rules for the Service Account, the Secret resource creates a similar authorization token as Pod with automounting of service account tokens would create and mount on the containers. When the token created by the Secret resource is mounted to the same path as the automatic mounting, as in example 3, the token is only mounted to the container that needs it. With this approach, the account tokens are only mounted into containers that actually need the them and the tokens cannot be used for privilege escalation from other containers of the Pod.

```
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      automountServiceAccountToken: false
      containers:
        - name: main
          image: main-image
          resources:
            limits:
              cpu: "2"
              memory: 1024Mi
          volumeMounts:
            - name: service-account
              mountPath: /var/run/secrets/kubernetes.io/serviceaccount
        - name: sidecar-container
          image: sidecar-image
          resources:
            limits:
              cpu: "1"
              memory: 512Mi
      volumes:
        - name: service-account
          secret:
            secretName: foo-service-account-token
```

Listing 3: Two container Pod with resource limits and manually mounted ServiceAccount

4.2.2 Enforcing the fields

There is no built-in enforcement tool for the fields in Kubernetes. However, Admission Controller can be used for enforcement by creating a custom ValidatingAdmission-

Webhook that fails Pod creation if the fields are not correctly set. This is actually how Pod Security Admission works under the hood: it is just built-in to the Kubernetes system as an original hook.

4.3 Solution

TODO: Summarize all the solutions into a one

5 Network Isolation

This chapter introduces solutions for implementing zero trust network architecture in K8s cluster. As disucssed in earlier chapters, the common network namespace and IP address prevents isolation of containers of the Pod. Thus, the isolation should either enforced with firewall rules inside the namespace, or the sidecar containers should be deployed in different network namespaces. The chapter provides solutions that implement both of these approaches.

5.1 Applying firewall rules inside Pod network namespace

5.1.1 Reserved source ports

The pair of source IP address and source ephemeral port would be an unique identifier for the communication; however, this would require application layer implementation to reserve ports per container.

5.1.2 IPTables

One possible solution for firewalling sidecar from the application is using IPTables.

Usually IPTables rules only refer to IP addresses and ports. However, IPTables ships with an owner module, which support rules based on application's user, group, process (TODO: check this) and session identifier.

Running IPTables in Pod context requires root user permissions with NET_ADMIN and NET_RAW capabilities.

The execution context of

- If executed from containers in Pod (init, lifecycle, wrapper container), it breaks Security admission rules (root user and NET_ADMIN)
- Can be executed from Node itself, using DaemonSet (sort of a CNI plugin), but a bit hacky.
- Use owner module for catching egress packets (userId, groupId, processId)

5.1.3 eBPF program firewall

- XDP works only for ingress
- TC needs some way to catch egress from sidecar

5.2 Own network namespace for sidecar

- Guaranteed to work, since own network namespaces. What type of issues arise from this?
- Need to force pods to same node, for common volumes (if using host as storage)

```
#!/bin/bash
sysctl -w net.ipv4.conf.all.route_localnet=1
iptables -t nat -A OUTPUT -p ALL --dport 8125 -j DNAT --to-destination 192.1
iptables -t nat -A POSTROUTING -j MASQUERADE
```

Listing 4: Script for mapping localhost to br0 interface

- Implementation by hand, or Admission controller that catches sidecars?
- Loopback is not the same anymore. DNAT that changes localhost to external?

5.2.1 Multus

- Requires breaking sidecar pattern with multiple Pods
- Same as own Pod for sidecars, but allows use of custom IP addresses
- Hard to implement between nodes, affinity rules
- Loopback DNAT

6 Solution Evaluation

7 Discussion

8 Conclusion

References

[1] AppArmor. *AppArmor*. 2022. URL: https://apparmor.net/ (visited on 03/31/2023).

[2] Docker Authors. *Docker overview*. 2023. URL: https://docs.docker.com/get-started/overview/ (visited on 04/06/2023).

[3] Docker Authors. *Use containers to Build, Share and Run your applications*. 2023. URL: https://www.docker.com/resources/what-container/ (visited on 04/06/2023).

[4] etcd Authors. *etcd*. 2023. URL: https://etcd.io/ (visited on 04/05/2023).

[5] Kubernetes Authors. *Admission Controllers Reference*. 2023. URL: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/ (visited on 04/06/2023).

[6] Kubernetes Authors. *Cluster Networking*. 2022. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/ (visited on 03/09/2023).

[7] Kubernetes Authors. *Deployments*. 2023. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visited on 04/04/2023).

[8] Kubernetes Authors. *Kubernetes*. 2023. URL: https://kubernetes.io/ (visited on 03/31/2023).

[9] Kubernetes Authors. *Kubernetes components*. 2023. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 04/05/2023).

[10] Kubernetes Authors. *Pod Security Admission*. 2023. URL: https://kubernetes.io/docs/concepts/security/pod-security-admission/ (visited on 04/26/2023).

[11] Kubernetes Authors. *Pod Security Standards*. 2023. URL: https://kubernetes.io/docs/concepts/security/pod-security-standards/ (visited on 04/04/2023).

[12] Kubernetes Authors. *Pods*. 2023. URL: https://kubernetes.io/concepts/workloads/pods/ (visited on 04/04/2023).

[13] Kubernetes Authors. *Resource Management for Pods and Containers*. 2023. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ (visited on 05/30/2023).

[14] Kubernetes Authors. *Services*. 2023. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 04/04/2023).

[15] The CNI Authors. *Container Network Interface (CNI) Specification*. 2023. URL: https://www.cni.dev/docs/spec/ (visited on 04/20/2023).

[16] Gerald Budigiri et al. “Network policies in kubernetes: Performance evaluation and security analysis”. In: *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. IEEE. 2021, pp. 407–412.

[17] Thanh Bui. “Analysis of docker security”. In: *arXiv preprint arXiv:1501.02967* (2015).

[18] Brendan Burns and David Oppenheimer. “Design patterns for container-based distributed systems”. In: *8th {USENIX} workshop on hot topics in cloud computing (HotCloud 16)*. 2016.

[19] Brendan Burns et al. “Borg, Omega, and Kubernetes”. In: *Communications of the ACM* 59.5 (2016), pp. 50–57.

[20] Cloudflare. *How to drop 10 million packets per second*. 2018. URL: https://blog.cloudflare.com/how-to-drop-10-million-packets/ (visited on 03/15/2023).

[21] Theo Combe, Antony Martin, and Roberto Di Pietro. “To docker or not to docker: A security perspective”. In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62.

[22] Cilium Developers. *Cilium*. 2023. URL: https://cilium.io/ (visited on 03/14/2023).

[23] Cilium Developers. *Component overview*. 2023. URL: https://docs.cilium.io/en/v1.13/overview/component-overview/ (visited on 04/12/2023).

[24] Cilium Developers. *Component overview*. 2023. URL: https://docs.cilium.io/en/v1.13/security/policy/language/ (visited on 04/12/2023).

[25] Cilium Developers. *Kubernetes Without kube-proxy*. 2023. URL: https://docs.cilium.io/en/v1.13/network/kubernetes/kubeproxy-free (visited on 04/12/2023).

[26] Cilium Developers. *Transparent Encryption*. 2023. URL: https://docs.cilium.io/en/v1.13/security/network/encryption/ (visited on 04/12/2023).

[27] Cilium developers. *Program types*. 2018. URL: https://docs.cilium.io/en/latest/bpf/progtypes/ (visited on 03/16/2023).

[28] Istio developers. *The Istio service mesh*. 2023. URL: https://istio.io/latest/about/service-mesh/ (visited on 04/06/2023).

[29] Linux Developers. *capabilities(7) — Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html (visited on 03/31/2023).

[30] Linux Developers. *namespaces(7) — Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 03/31/2023).

[31] Linux Developers. *seccomp(2) — Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man2/seccomp.2.html (visited on 03/31/2023).

[32] Docker. *Docker default Seccomp profile*. 2022. URL: https://github.com/moby/moby/blob/23.0/profiles/seccomp/default.json (visited on 03/31/2023).

[33] Docker. *Docker overview*. 2018. URL: https://docs.docker.com/get-started/overview/ (visited on 03/16/2023).

[34] Docker. *Docker security*. 2023. URL: https://docs.docker.com/engine/security/ (visited on 03/31/2023).

[35] Docker. *libnetwork*. 2023. URL: https://github.com/moby/moby/tree/master/libnetwork (visited on 03/10/2023).

[36] Martin Fowler and James Lewis. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 05/09/2023).

[37] Thomas Graf. *Cilium Service Mesh – Everything You Need to Know*. 2023. URL: https://isovalent.com/blog/post/cilium-service-mesh/ (visited on 04/12/2023).

[38] Network Plumbing Working Group. *Multus-CNI*. 2023. URL: https://github.com/k8snetworkplumbingwg/multus-cni (visited on 04/25/2023).

[39] Network Plumbing Working Group. *Multus-CNI Configuration Reference*. 2023. URL: https://github.com/k8snetworkplumbingwg/multus-cni/blob/master/docs/configuration.md (visited on 04/25/2023).

[40] The Kubernetes Networking Guide. *Calico*. 2023. URL: https://www.tkng.io/cni/calico/ (visited on 03/14/2023).

[41] The Kubernetes Networking Guide. *Cilium*. 2023. URL: https://www.tkng.io/cni/cilium/ (visited on 03/14/2023).

[42] The Kubernetes Networking Guide. *CNI*. 2023. URL: https://www.tkng.io/cni/ (visited on 04/20/2023).

[43] Michael Hausenblas. *Container Networking*. O'Reilly Media, Incorporated, 2018.

[44] Toke Høiland-Jørgensen et al. "The express data path: Fast programmable packet processing in the operating system kernel". In: *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 2018, pp. 54–66.

[45] Pooyan Jamshidi et al. "Microservices: The journey so far and challenges ahead". In: *IEEE Software* 35.3 (2018), pp. 24–35.

[46] Alper Kerman et al. "Implementing a zero trust architecture". In: *National Institute of Standards and Technology (NIST)* (2020).

[47] Xin Lin et al. “A measurement study on linux container security: Attacks and countermeasures”. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. 2018, pp. 418–429.

[48] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture.” In: *USENIX winter*. Vol. 46. 1993.

[49] Dirk Merkel et al. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux j* 239.2 (2014), p. 2.

[50] Sebastiano Miano et al. “A framework for eBPF-based network functions in an era of microservices”. In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 133–151.

[51] Microsoft. *Threat Matrix for Kubernetes*. 2023. URL: https://microsoft.github.io/Threat-Matrix-for-Kubernetes/ (visited on 05/24/2023).

[52] Shixiong Qi, Sameer G Kulkarni, and KK Ramakrishnan. “Assessing container network interface plugins: Functionality, performance, and scalability”. In: *IEEE Transactions on Network and Service Management* 18.1 (2020), pp. 656–671.

[53] Scott Rose et al. *Zero trust architecture*. Tech. rep. National Institute of Standards and Technology, 2020.

[54] Nataliya Shevchenko et al. *Threat modeling: a summary of available methods*. Tech. rep. Carnegie Mellon University Software Engineering Institute Pittsburgh United . . ., 2018.

[55] Tigera. *About Calico*. 2023. URL: https://docs.tigera.io/calico/3.25/about (visited on 04/13/2023).

[56] Tigera. *About Network Policy*. 2023. URL: https://docs.tigera.io/calico/latest/about/about-network-policy (visited on 04/13/2023).

[57] Tigera. *Install calicoctl*. 2023. URL: https://docs.tigera.io/calico/3.25/operations/calicoctl/install (visited on 04/13/2023).

[58] Marcos AM Vieira et al. “Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications”. In: *ACM Computing Surveys (CSUR)* 53.1 (2020), pp. 1–36.

A Dockerfile for penetration testing

```
FROM --platform=linux/amd64 ubuntu:22.04

RUN apt update
RUN apt install -y curl

# Setup kubectl
RUN curl -fsSLo /etc/apt/keyrings/kubernetes-archive-keyring.gpg https://package
RUN echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-archive-keyring.gpg] https

# Install deps
RUN apt update
RUN apt install -y net-tools nmap ncat kubectl etcd iputils-ping iproute2

# kubeletctl
RUN curl -LO https://github.com/cyberark/kubeletctl/releases/download/v1.9/kubel

RUN apt install -y iptables

# Point to the internal API server hostname
RUN echo 'export APISERVER=https://kubernetes.default.svc' >> root/.bashrc
# Path to ServiceAccount token
RUN echo 'export SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount' >
# Read this Pod's namespace
RUN echo 'export NAMESPACE="$(cat ${SERVICEACCOUNT}/namespace)"' >> root/.bashrc
# Read the ServiceAccount bearer token
RUN echo 'export TOKEN="$(cat ${SERVICEACCOUNT}/token)"' >> root/.bashrc
# Reference the internal certificate authority (CA)
RUN echo 'export CACERT="${SERVICEACCOUNT}/ca.crt"' >> root/.bashrc
```

Listing 5: Dockerfile for penetration testing

B Example webserver deployed with StatsD sidecar

```
const Koa = require('koa');
const app = new Koa();

const StatsD = require('node-statsd');
const client = new StatsD();

const logMessage = async (ctx, next) => {
  await next();
  const rt = ctx.response.get('X-Response-Time');
  console.log(`${ctx.method} ${ctx.url} - ${rt}`);
  client.timing('response_time', rt);
  client.increment('response_counter');
}

const setResponseTimeCtx =async (ctx, next) => {
  const start = Date.now();
  await next();
  const ms = Date.now() - start;
  ctx.set('X-Response-Time', `${ms}ms`);
}

app.use(logMessage);
app.use(setResponseTimeCtx);
app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(8888);
```

Listing 6: Node.js server (`index.js`)

```
  {
  "name": "node-app",
  "version": "1.0.0",
  "description": "Webserver w/ StatsD",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Aarni Halinen",
  "license": "ISC",
  "dependencies": {
    "koa": "^2.14.2",
    "node-statsd": "^0.1.1"
  }
}
```

Listing 7: `package.json`

```
FROM node:18-alpine

WORKDIR /app

COPY index.js package.json package-lock.json ./
RUN npm install --omit=dev

ENTRYPOINT [ "npm", "start" ]
```

Listing 8: Dockerfile for the server

C Webserver and StatsD deployed with Multus networking

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-app
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node-app
  template:
    metadata:
      labels:
        app: node-app
      annotations:
        k8s.v1.cni.cncf.io/networks: '[{
            "name": "localhost-replacement",
            "interface": "br0",
            "ips": [ "192.168.1.201/24" ]
          }]'
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                - minikube-m02
      automountServiceAccountToken: false
      containers:
        - name: node-app
          image: node-app
          command: [ "node" ]
          args: [ "index.js" ]
          ports:
            - containerPort: 8888
              protocol: TCP
          securityContext:
            allowPrivilegeEscalation: true
            readOnlyRootFilesystem: false
            runAsUser: 0
            runAsGroup: 0
            runAsNonRoot: false
            seccompProfile:
              type: RuntimeDefault
            capabilities:
              drop:
                - ALL
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-app-statsd
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node-app-statsd
  template:
    metadata:
      labels:
        app: node-app-statsd
      annotations:
        k8s.v1.cni.cncf.io/networks: '[{
            "name": "localhost-replacement",
            "interface": "br0",
            "ips": [ "192.168.1.202/24" ]
          }]'
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                - minikube-m02
      automountServiceAccountToken: false
      containers:
        - name: statsd
          # image: statsd/statsd
          image: hypnza/statsd_dumpmessages
          ports:
            - containerPort: 8125
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            runAsUser: 101
            runAsGroup: 101
            runAsNonRoot: true
            seccompProfile:
              type: RuntimeDefault
            capabilities:
              drop:
                - ALL
          resources:
            limits:
```

```
  apiVersion: "k8s.cni.cncf.io/v1"
  kind: NetworkAttachmentDefinition
  metadata:
    name: localhost-replacement
    namespace: app
  spec:
    config: '{
        "cniVersion": "0.3.1",
        "name": "localhost-replacement",
        "capabilities": { "ips": true },
        "type": "macvlan",
        "master": "eth0",
        "mode": "bridge",
        "ipam": {
          "type": "static"
        }
      }'
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: app
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
---
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: deny-all-br0
  namespace: app
  annotations:
    k8s.v1.cni.cncf.io/policy-for: localhost-replacement
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
---
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: allow-statsd
  namespace: app
  annotations:
    k8s.v1.cni.cncf.io/policy-for: localhost-replacement
spec:
```