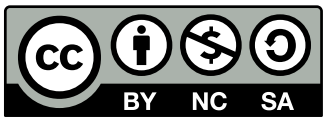
Aalto University
School of Science

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** 55+9 **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, the abstract should be written in both 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 and in the pdf file's metadata via the thesisabstract macro (see the comment in the TeX file). Write in this 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 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** 55+9 **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

Abstract 3

Abstract (in Finnish) 4

Preface 5

Contents 6

Abbreviations 8

1 Introduction 9
- 1.1 Problem Statement . . . 9
- 1.2 Thesis outline . . . 10

2 Background 11
- 2.1 Zero trust architecture . . . 11
- 2.2 Containerization and Docker . . . 11
  - 2.2.1 Linux containers . . . 12
  - 2.2.2 Docker . . . 13
- 2.3 Kubernetes . . . 14
  - 2.3.1 Kubernetes objects . . . 14
  - 2.3.2 Kubernetes components . . . 16
  - 2.3.3 Admission controllers . . . 17
  - 2.3.4 Sidecar pattern in Kubernetes . . . 18
- 2.4 Kubernetes network model . . . 19
  - 2.4.1 Container Network Interface . . . 20
  - 2.4.2 Calico . . . 22
  - 2.4.3 Cilium . . . 23
  - 2.4.4 Multus . . . 24
  - 2.4.5 Extended Berkeley Packet Filter . . . 25

3 The threat model 27
- 3.1 Existing models . . . 27
- 3.2 Attacker model . . . 27

4 Solution requirements 32
- 4.1 Security requirements . . . 32
- 4.2 Environment . . . 32

5 Hardening Pod security 33
- 5.1 Restricted Pod Security Standard . . . 33
- 5.2 Enforcing other best practices . . . 35
  - 5.2.1 Manual service account mounting . . . 35
  - 5.2.2 Enforcing the fields . . . 37

5.3 Solution 38

6 Network Isolation **39**
6.1 Applying firewall rules inside Pod network namespace 39
6.1.1 IPTables 40
6.1.2 Deployment 43
6.2 Own network namespace for sidecar 43
6.2.1 Creating isolated network for sidecar traffic 44
6.2.2 Mutating sidecars to multiple Pods? 47

7 Solution Evaluation **48**

8 Discussion **49**

9 Conclusion **50**

References **51**

A Dockerfile for penetration testing **56**

B Example webserver deployed with StatsD sidecar **57**

C Webserver and StatsD deployed with Multus networking **59**

Abbreviations

API	Application Programming Interface
BGP	Border Gateway Protocol
BPF, cBPF	(classic) Berkeley Packet Filter
cgroups	Control groups
CLI	Command-line interface
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CRD	Custom Resource Definition
DAC	Discretionary Access Control
DoS	Denial of Service
eBPF	Extended Berkeley Packet Filter
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
IPAM	IP Address Management
IPC	Inter-process communication
K8s	Kubernetes
LXC	Linux Containers
MAC	Mandatory Access Control
NAT	Network Address Translation
NIC	Network Interface Controller
OOM	Out-of-memory
OpenVZ	Open Virtuozzo
OS	Operating System
PID	Process ID
RCE	Remote code execution
SELinux	Security-Enhanced Linux
Seccomp	Secure computing mode
TC	Traffic Control
VXLAN	Virtual Extensible LAN
XDP	eXpress Data Path
ZTA	Zero Trust Architecture

1 Introduction

During the last decade, the IT industry has shifted from monolithic software applications towards microservices. In the microservice architecture, each application is split into a suite of smaller independent services that handle part of the business logic [38]. Each service runs its own process, and the application data is sent between the components using lightweight communication mechanisms such as Hypertext Transfer Protocol (HTTP) and Google Remote Procedure Call (gRPC). This architectural approach also increases software agility because each service becomes an independent unit of development, deployment, operations, versioning, and scaling [51]. This modularity is often associated with benefits like faster delivery and improved scalability. JB: The point is that microservices have become popular because of containers During the same period of time, containers have become a widely used method to deploy 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 [19]. As such, a single microservice is quite often built and deployed as a container. In more complex systems with multiple containers, an orchestrator is often used to manage workloads. Kubernetes is one of the most widely used orchestrator tools. JB: Describe shortly what an orchestrator does and what is the difference between orchestrator and container runtime. Maybe this should be in the background.

Similarly to how design patterns emerged from the birth of object-oriented software systems, the modularity of containers and microservices has allowed the development of distributed system design patterns [20]. The most common of these patterns is the sidecar pattern, in which peripheral tasks like logging and observability are split away as their own containers from the main application container. The pattern allows easier installation of these nonfunctional features, while also keeping them away from the main container's source code. Basically, the 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 containment 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 one deployable unit.

1.1 Problem Statement

Although 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 sidecar tasks. The source code of these sidecar containers is not always available and finding vulnerabilities therein is not a trivial task. Furthermore, malicious actors can use supply chain attacks and typo-squatting to trick victims into installing malicious sidecars to their clusters. Once malicious attackers gain access to the sidecar, any misconfigurations 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 in permissive settings or outright disabled.

In Kubernetes, there is 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 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. JB: Statements are too weak. Here, the reader should understand that sidecars may contain vulnerable (or potentially vulnerable) code, and there are NO mechanisms to protect containers running in the same Pod.

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 paradigms, the capabilities of an individual component are limited to only those that are required for the component to function. The capabilities, such as 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 4 proposes ideas for isolating sidecars from the main application container. The chapter discusses both container and network security in the context of Kubernetes Pod. Chapter 7 introduces an implementation based on the findings of the previous chapter. The pros and cons of the solution are discussed in Chapter 8. Finally, Chapter 9 discusses future research and concludes the thesis.

2 Background

2.1 Zero trust architecture

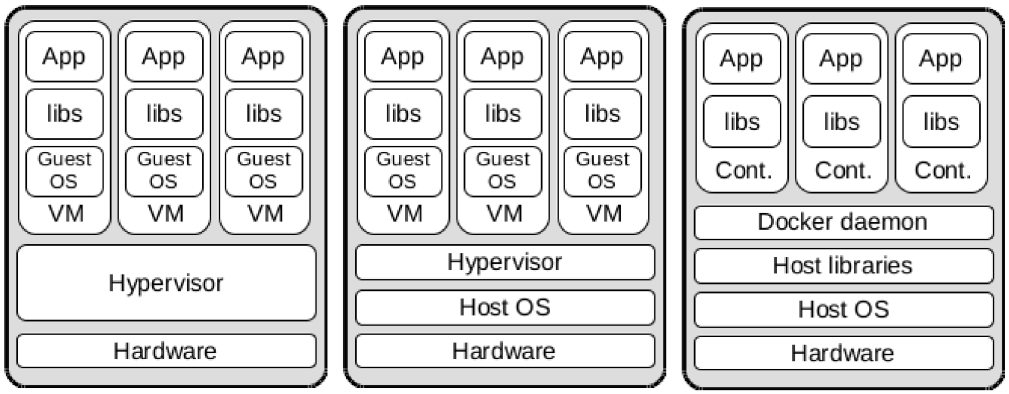
Conventional network security has historically focused on perimeter defense [52]. Subjects like workload resources and users inside the perimeter are often assumed to be trusted and implicitly given access inside the network, while any request originating outside the network is subject to more scrutiny. Although the systems seem initially secure, the modern IT landscape with cloud-based systems, third-party components, remote workers, etc. increases the attack surface of threat actors. 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 continuously evaluated [52, 61]. In contrast to a single perimeter defense, the focus in ZTA is to create 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 the number of trusted subjects equals 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 within the perimeter, malicious actors could only laterally move to the resources that the compromised component required to function. Any other component is still protected by its own perimeter, which 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. JB: How is the Zero trust implemented in practice? Is there a relationship between zero trust and sidecars?

2.2 Containerization and Docker

Figure 1 illustrates common virtualization models. While traditional virtualization techniques virtualize workloads on top of a hypervisor that shares hardware resources between virtual machines, containerization is a technique where virtualization occurs at the operating system level [55]. Processes executing in containers run on the kernel of the host machine. 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 [22]. 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 [36] is a hugely



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

Figure 1: Virtualization models [22]

popular LXC-based container runtime and provides an easy-to-use API and tooling 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 containment using a Linux kernel feature called namespaces [53]. Namespaces [32] are a construct that wraps a global system resource in an abstraction that makes it appear to the processes in the namespace that they have their own isolated instance of the global resource. There are a total of eight namespaces: i) *Cgroup* which is used for resource management, ii) *Inter-process communication* (IPC) that 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, the *network* namespace provides each container with its own loopback device, and even `iptables` rules. In another example, *mount* namespace ensures that container has no visibility or access to the host's or other container's file system. Compared to other namespaces that concern the isolation of kernel data, *cgroups* focuses on limiting available system resources per namespace [53]. Each namespace can be configured with its 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 the 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 security mechanisms are adopted from the Linux kernel to restrict the capabilities of containers [53]. The

mechanisms include Discretionary Access Control (DAC) mechanisms like Capability [31] and Secure computing mode (Seccomp) [33], and Mandatory Access Control (MAC) mechanisms such as 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 represents a permission to process some specific kernel resources. The feature turns the binary "root/non-root" security mechanism into a fine-grained access control system, which makes it easier to follow the principle of least privilege. For example, processes like web servers that simply need to bind on a Internet domain privileged port (numbers below 1024) do not need to run as root; they can be granted with `CAP_NET_BIND_SERVICE` capability instead [37]. The Seccomp mechanism constrains which system calls a process can invoke. The available system calls are defined for a container through the Seccomp profile which is defined as a JSON file. The default Docker Seccomp profile [35] includes more than 300 system calls. SELinux is integrated into 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 [53].

2.2.2 Docker

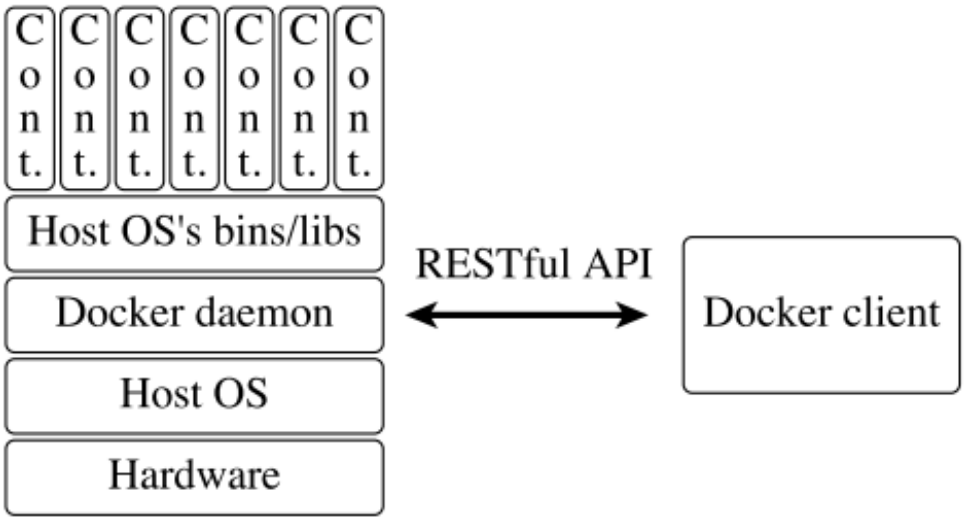


Figure 2: Architecture of Docker engine [19]

Docker is an open-source container technology written in Go and launched in 2013 [3, text]. The platform consists of the Docker Engine packaging tool, Docker image registries such as the Docker Hub public image repository and the Docker desktop application [2]. In general, the engine architecture is similar to container-based virtualization, as visible in Figure 2 [19]. The containers run on top of the Docker daemon, which manages and executes all the containers. The daemon is exposed to

Docker clients via the 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 the daemon with the client. Remote communication with the API should be carried out over TLS for security reasons.

Docker image is a read-only template with instructions for creating a Docker container [2]. Images are often based on another image, such as OS images `ubuntu` and `alpine`, with some additional customizations, such as the installation of web server binaries. Customizations are added to the image as a series of data layers so that each new command creates a new layer. This process makes image distribution more efficient since only the changes between layers must be distributed [19]. Layering is achieved with a special file system inspired by UnionFS that 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 proprietary 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 attempts to search and pull the image from the connected registries. JB: Docker can only create and destroy containers. This makes it a runtime. This is an important concept to mention.

2.3 Kubernetes

Kubernetes (K8s) [9] is an open-source container orchestrator, i.e., a system to automate the deployment, scaling and management of containerized applications. It allows the 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 attempts to restart and reschedule containers so that the desired cluster state is maintained. Kubernetes is hosted by the Cloud Native Computing Foundation (CNCF), but its origins are at Google, where it was created as an open-source option for Google's proprietary Borg and Omega orchestrators [21]. Kubernetes was open-sourced in 2014.

2.3.1 Kubernetes objects

Pods are the basic atomic scheduling unit in Kubernetes. Pods consist of one or more tightly-coupled containers with shared storage volume and networking [14]. Containers in a Pod are always co-located and co-scheduled and run in a shared context, i.e. a set of Linux namespaces. The network, UTS and IPC namespaces are shared by default, and the 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 a common IP address, and cannot reuse the same port numbers. In addition to the normal application container, Pods can include special `initContainers` that are run only on Pod startup. These Pods are used to modify

the 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.

If a Pod fails, a replacement Pod is not automatically created. Quite often, developers want to have more control over the compute resources and specify a target state for them. This includes replication, scaling, and distribution among various nodes. To meet these needs, Kubernetes provides additional resources.

Instead of directly creating Pods, **Deployment** workload resources can be used to create Pods in a cluster [14]. With Deployments, the 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 availability of the desired number of Pods [8]. **DaemonSet** on the other hand is a workload resource that ensures that all or some Nodes run a copy of a Pod. Typical use cases for daemons are running Node monitoring and logging, and network plugins, which are discussed in depth in the Chapter 2.4.1.

All Pods across the cluster share the same subnet and can access each other via IP address. However, connecting to a Pod with IP address is suboptimal, since Pods are ephemeral. A dead Pod, even if controlled by a ReplicaSet, is not guaranteed to receive the same IP address on restart. Additionally, each replica in a horizontally scaled system has its own IP address. This leads to a problem: how do the clients using the system find and keep track of the IP addresses used by the workload? The **Service** abstraction solves the problem.

Services are an object for exposing groups of Pods over a network [16]. The object defines a set of endpoints, that is, 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`: i) the default `ClusterIP` which exposes the Service inside the cluster with its own IP address, ii) `NodePort` which exposes the 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.

JB: Usually services are proxies that route requests to the different Pods. There are also Headless services that only add DNS entries for the Pods.

Namespaces allow logical grouping of resources under a single name. New Kubernetes 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 Kubernetes system which is discussed in more detail in the next Chapter 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 [13], which are used by the Pod Security admission

controller, are also defined at the namespace level. Admission controllers are discussed in Chapter 2.3.3.

Custom Resource Definitions (CRD) are used to define new resources that are not available in a default Kubernetes installation [7]. Once a custom resource is installed, users can create and access custom resource objects with `kubectl`, similarly to any other built-in resource. On their own, custom resources can only be used to store and retrieve structured data. When combined with a custom controller, custom resources can be used to add new functionality to the cluster. In Kubernetes, controllers are a construct that watch the current state of the cluster and keep track of the desired state. If the states differ, the controller makes or requests changes so that the cluster state moves closer to the desired one. Specifically, the controllers are implemented as **Operators** following the *operator-pattern* [11]. The Operators are clients of the Kubernetes API that implement the control loop on a Custom Resource. They are often deployed as Deployments, and behave similarly to any other container workload in the cluster.

2.3.2 Kubernetes components

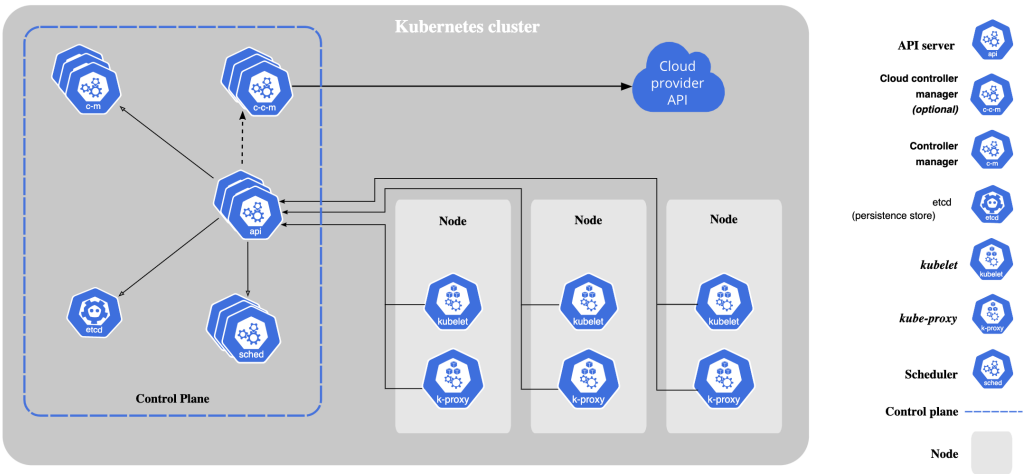


Figure 3: Kubernetes cluster architecture [10]

Figure 3 describes the Kubernetes cluster with a 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 the complete and working Kubernetes cluster [10]. The control plane components can run on any worker node. However, clusters often have a 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 front-end component of the control plane. It is an HTTP server that is used to send commands to the cluster. The server handles authentication and validation of the commands. For valid commands, the server then forwards these to

other control plane components that then modify the cluster state. The easiest way to send commands to the server is by using the `kubectl` command-line interface (CLI) tool, which actively sends the commands as HTTP under the hood. 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 highly 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 automatically 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 consistent across all instances. On leader failure, the other instances automatically elect a 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 [10]. Scheduling takes into account resource availability 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 [10]. Controller loops, like the Deployment controller, continuously 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 ensures that containers are running in a Pod [10]. 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. `Kubelet` only manages containers created by Kubernetes. **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 traffic [27].

2.3.3 Admission controllers

Admission controllers are a feature of the Kubernetes API server, used to validate and modify requests made to the server [5]. The controllers execute before the request is executed but after it is authenticated and authorized by the server. Several important features of Kubernetes are implemented with admission controllers, and these should be enabled on 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 first executes the mutating controllers so that no mutations occur 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.

One notable Admission controller is the `PodSecurity` controller. The controller validates the Pods before they are admitted, making sure that the requested Pod security context and other restrictions are permitted in the namespace to which the Pod is assigned [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 LEVEL is a predefined level of the Pod Security Standard. The three available levels are `privileged`, `baseline` and `restricted` [13].

The available actions 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 [12]. 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 violation of security policies without blocking the request by setting the `warn` mode more restrictive than `enforce`.

2.3.4 Sidecar pattern in Kubernetes

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 [20]. The 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 paired with a container that collects server logs from a file and streams them to a centralized log management system. Listing 1 shows how the logs can be shared to the sidecar with file mounts. Another example of this pattern is the Istio service mesh [30] and its Envoy proxy sidecar, which routes all traffic through the Istio control plane for management, observability, and security operations.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-logs
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: nginx-logs
10   template:
11     metadata:
12       labels:
13         app: nginx-logs
14     spec:
```

```
15      containers:
16        - name: main-application
17          image: nginx
18          volumeMounts:
19            - name: shared-logs
20              mountPath: /var/log/nginx
21        - name: sidecar-container
22          image: busybox
23          command: ["sh","-c","while true; do cat /var/
    log/nginx/access.log; sleep 30; done"]
24          volumeMounts:
25            - name: shared-logs
26              mountPath: /var/log/nginx
27      volumes:
28        - name: shared-logs
29          emptyDir: {}
```

Listing 1: Nginx web server with a sidecar that periodically reads the logs

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. Although sidecar functionality could be built into the main container, there are benefits in using separate containers [20]. The isolation allows tweaking of containers' cgroups so that CPU cycles can be prioritized for the main container. The isolation also provides a failure containment boundary between the main and sidecar processes. Since the container is also the unit of deployment, sidecar containers could be developed, tested, and deployed independently of each other. Sidecar containers can also be developed with different tools and dependencies and in a way that they can be reused with other application containers. From the testing point of view, the componentized system might improve testing if the smaller units can be tested independently. However, on the downside, the combination of all container version combinations that might be seen in a production environment also increases.

2.4 Kubernetes network model

A fundamental part of a Kubernetes cluster is how nodes and resources are networked together. Specifically, the networking model needs to address four different types of networking problems: i) intra-Pod (that is, container-to-container within the 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 [59]. All agents on a host should also be able to communicate with Pods on the same host. Kubernetes does not implement this model out-of-the-box, but hands the implementation to special Container Network Interface (CNI) plugins.

This allows different vendors and operators to use varying networking mechanisms in Kubernetes.

2.4.1 Container Network Interface

The Container Network Interface [17] is a networking specification, which has become the de-facto industry standard for container networking and is backed by CNCF [59]. The specification was first developed for the container runtime `rkt` [48], but it is now supported by nearly all major container runtimes and orchestrators. As many runtimes seek to solve the same problem of making the network layer pluggable, the CNI was developed as a common API to promote interchangeability. Most container orchestrators have adopted the specification as their networking solution. The biggest outlier is Docker Swarm, which instead implements its own proprietary approach to container networking.

The CNI specification has five distinct definitions: i) a format for network configuration, ii) an 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 delegating functionality between the plugins, and v) data types for the plugins to return their results to the runtime [17]. 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 the 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, is passed to the plugins via the 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 runtime mutates a container network, it results in a series of ADD, DELETE, or CHECK executions. These are then executed in the 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 [47]. For connectivity, each Pod must have a network interface controller (NIC) for communication outside its networking namespace. The NIC must have an IP address that is 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 should receive a unique IP address from an IP pool range designated for the Pods. When a cluster is installed, the administrator assigns a CIDR for the whole cluster. Then the `kube-controller-manager` can be configured to assign each node its own CIDR range, defined in the Node's `spec.podCIDR` field. The IP addresses are assigned to the Pods by an IP address management (IPAM)

plugin. IPAM plugins are *delegated plugins* of the CNI, which means that the CNI plugin is responsible for invoking the IPAM plugin when needed. Thus, many CNI plugins are installed with their own IPAM plugins for convenience. Quite often IPAM plugins assign Pods with IP address from the Node's `podCIDR`, but sophisticated IPAMs like those of Calico and Cilium use Custom Resources for more configurable IP pools. The end-to-end reachability between different Node `PodCIDR`s is established by encapsulating in the overlay network, for example, with Virtual Extensible LAN (VXLAN), or orchestrating on the underlay network, e.g. with Border Gateway Protocol (BGP).

Since Kubernetes does not provide networking between Pods, it has no capabilities to enforce network isolation between workloads. Thus, another key feature of some CNI plugins is the enforcement of network traffic rules. For this purpose, Kubernetes provides a common built-in resource called `NetworkPolicy` for the CNI plugins to consume. The Listing 2 is an example of network policies for a web application front-end. The first policy functions as a default rule that denies all implicit egress and ingress traffic. The second policy allows traffic to the default HTTP port, and the third policy allows the front-end application to send traffic to any Pod in the *backend* namespace.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: frontend
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webapp-ingress
  namespace: frontend
spec:
  podSelector:
    matchLabels:
      app: webapp
  policyTypes:
    - Ingress
  ingress:
    - ports:
        - protocol: TCP
          port: 80
```

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webapp-egress
  namespace: frontend
spec:
  podSelector:
    matchLabels:
      app: webapp
  policyTypes:
    - Egress
  egress:
    - to:
        - namespaceSelector:
          matchExpressions:
            - key: namespace
              operator: In
              values: ["backend"]
```

Listing 2: Example NetworkPolicies for a frontend web application

The NetworkPolicy specification consists of a `podSelector` that specifies Pods that are subject to the policy and `policyTypes` to specify the Ingress and Egress rules for the traffic [18] 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.

Although 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 [63] is an open-source CNI plugin with modular architecture that supports a wide range of deployment options. Each Pod created in the Calico network receives one end of a virtual ethernet device link as its default `eth0` network interface, while the other end is left dangling on the host Node [45]. The Pod end of the link receives an IP address from Pod CIDR, but the Node end does not. Instead, a `proxy_arp` flag is set 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 its own CLI tool, called `calicoctl` [65], to manage 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. 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, and more flexible matching criteria (e.g., mathcing on ServiceAccounts) [64]. The policy can also match other Calico CRDs such as **HostEndpoints** and **NetworkSets**, which allows implementing rules on host interfaces and non-Kubernetes resources. If Calico is installed along the 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.
JB: Now that Calico uses eBPF there is little difference between them and both can reduce the amount of sidecars to one per node.

2.4.3 Cilium

Cilium [24] is one of the most advanced and powerful CNI plugins for Kubernetes. Similarly to Calico, it creates a virtual ethernet device for each Pod and sets one side of the link into the Pod's network namespace [46] as the default interface. Cilium then attaches extended Berkeley Packet Filter (eBPF) programs to ingress traffic control (`tc`) hooks of these virtual ethernet devices to intercept 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 [18]. 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 the BGP protocol as well [24].

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`) [25]. The agent daemons subscribe to events from the 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 one to inspect the state and status of the local agent. The tool should not be confused with the Cilium management CLI tool, also incidentally named `cilium`, which is typically installed remote from

the cluster. The operator is responsible for all management operations that should be handled once for the entire cluster, rather than once for each Node. This includes, for example, the registration of CRDs.

Although default Kubernetes NetworkPolicies provides security on OSI layers L3 and L4, Cilium provides CRDs that also support L7 policies [26]. 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 all namespaces 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 [27]. With `kubeProxyReplacement` installation option, Cilium implements Service load-balancing as XDP and TC programs on the Node network stack. For encryption, Cilium supports IPsec and WireGuard implementations [28]. The service mesh performs a variety of features directly in eBPF, thus functioning without sidecar containers or proxying requests through the agent Pod's Envoy [39]. Since all features are not available as eBPF programs or on all kernel versions, Cilium automatically probes the underlying kernel and automatically reverts to the Envoy proxy when needed. For capabilities beyond the built-in mesh, Cilium also provides integration with Istio.
JB: Have you thought about how your solution is impacted by eBPF?

2.4.4 Multus

Traditionally, CNI plugins provide only a single network interface for a Pod, apart from the loopback device. Multus [43] is a CNI plugin that allows the attachment of multiple network interfaces for a Pod. It does not provide any connectivity or reachability for containers like other plugins. Instead, it is installed as the first plugin in the CNI plugin chain. When executed, the plugin delegates the creation of the interface 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 on 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 configures the configuration file and optionally spawns an HTTP server for additional features such as metrics [43]. The configuration file satisfies the CNI specification with few additional attributes of which the combination of `clusterNetwork` and `defaultNetworks` or `delegates` are imperative for the CNI plugin to function [44].

The `clusterNetwork` specifies the main network of the cluster, which implements the `eth0` interface and the 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 the 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 inferred 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 the `NetworkAttachmentDefinition` names as input. The network interface identifiers can be modified by providing 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 the JSON array format.

2.4.5 Extended Berkeley Packet Filter

Berkeley Packet Filter (BPF, or nowadays often cBPF) was originally developed in the early 1990s as a high-performance tool for user-space packet capture [54]. 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 instructions 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 use-case for BPF is *libpcap* library, which is used by a 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 [66]. This reworked version of BPF was first merged into the Linux kernel in 2014 and is publicly called the 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 into the kernel with `bpf()` system call [56]. Most often, programs are written in restricted C and compiled with the LLVM Clang compiler to bytecode. It is also possible to use the eBPF assembly instructions and `bpf_asm` utility for converting instructions to bytecode. eBPF programs follow an event-driven architecture: a loaded eBPF program is hooked to a particular type of event and each occurrence of the event triggers the program execution.

There are two different network event interfaces in eBPF: eXpress Data Path

(XDP) and Traffic Control (TC) [56]. XDP programs are attached to an NIC and can handle only incoming packets [49]. The programs are called directly by the NIC driver if it has XDP support, thus executing before packets enter the network stack. This skips expensive packet parsing and memory allocation operations and allows XDP programs to run at very high throughput. Thus, even the main network buffer *skbuff* is not populated. Some SmartNICs even support offloading the program to the NIC's own processor from the host CPU, further improving the performance of the host machine [29]. 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 the contents of packets [66]. Since the packets are not parsed into the network stack, the programs have to work with raw packets and implement their 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 on which it arrived with `XDP_TX`, usually after modifying the contents of the packet. `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 the kernel traffic control function within the Linux network stack [66]. The ingress hook runs 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 a packet in memory. Similarly to XDP programs, the return value of the program determines the further processing of the packet. The packet can be passed further 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. JB: Do we need this level of detail to understand your solution?

3 The threat model

This chapter analyzes threats in the Kubernetes cluster from the perspective of sidecars. The chapter first discusses existing Kubernetes threat models, which are then used to identify possible attack vectors and readily available mitigations. Finally, the chapter discusses the attack model and demonstrates attacker capabilities with example attack scenarios.

3.1 Existing models

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 potential threats to a system. This includes profiles of potential attackers and their goals and methods, as well as potential mitigations [62].

For Kubernetes clusters, there exists an extensive *Threat Matrix for Kubernetes* created by Microsoft [57]. The matrix is adapted from *MITRE adversarial tactics, techniques, and common knowledge (ATT&CK)* Framework's container matrix [23] and is a de-facto industry standard for describing threats. The matrix, illustrated in Figure 4, describes common techniques used by attackers in chronological stages, from initial access to impact [58]. A defense-in-depth strategy is achieved by addressing threats at all stages of attack.

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Collection	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API server	Access cloud resources	Images from a private registry	Data Destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account		Resource Hijacking
Kubeconfig file	New container	Kubernetes CronJob	hostPath mount	Pod / container name similarity	Access container service account	Network mapping	Cluster internal networking		Denial of service
Application vulnerability	Application exploit (RCE)	Malicious admission controller	Access cloud resources	Connect from Proxy server	Applications credentials in configuration files	Instance Metadata API	Applications credentials in configuration files		
Exposed sensitive interfaces	SSH server running inside container				Access managed identity credential		Writable volume mounts on the host		
	Sidecar injection				Malicious admission controller		CoreDNS poisoning		
							ARP poisoning and IP spoofing		

Figure 4: Kubernetes threat matrix. The attack techniques addressed are highlighted in green.

3.2 Attacker model

The adversary in this model is a compromised sidecar container in the cluster. Since sidecar containers do not technically differ from main containers, they are compromised

in a similar fashion. For example, the cluster could run a container with security vulnerability or the admins could have been tricked to installing a malicious sidecar to the cluster. Thus, the model assumes that the first stage is already breached. The model also assumes that the adversary has execution capabilities in the sidecar container, specifically a shell access to the sidecar container in the examples. This type of security breach can occur if the application has a remote code execution (RCE) vulnerability that the attacker can exploit to run a reverse shell script inside the application. As another example, the attacker could acquire valid credentials to an application running a Secure Shell (SSH) server through brute-force or phishing attacks. The goal of the model is to prevent the attacker from escalating the attack from the initial breach. Tables 1 and 2 list the identified threats, with threats related to networking divided into the latter.

Table 1: Kubernetes sidecar threats

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 their own namespaces to follow the principle of least privilege.
Writing to the host file system	TODO.	Do not allow 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 needed. Adhere to the principle of least privilege.
There are no resource limits for containers	If not set, a single container can hog system resources, causing denial of service (DoS).	Set resource limits to containers[3].

[1] `spec.serviceAccountName` is a Pod-level field
[2] Set `automountServiceAccountToken: false`
[3] Use `spec.containers[*].resources`

Table 2: Networking threats for sidecars

Threat	Description	Mitigation
The Pod has network access to the control plane	By default, all traffic inside the cluster is allowed.	Use NetworkPolicies as a firewall. Deny by default and explicitly allow traffic only if needed.
Sidecar can use the main container's NetworkPolicy to access `kube-system` or other Pods	NetworkPolicies affect the NIC of the Pod network, which is shared by sidecars.	**No built-in solution available!**
Sidecar has network access to the main container	Main container is accessible via loopback device, which cannot be protected with NetworkPolicy.	**There is no built-in solution available!**
Sidecar sniffs the main application traffic	Pod-to-Pod traffic in cluster is unencrypted by default. A privileged sidecar can sniff and hijack the traffic.	CNIs and service meshes may provide encryption to inter-Pod traffic with, for example, IPSec, Wireguard and mutual TLS. For intra-Pod traffic, the only feasible solution is to limit the container permissions. (TODO: TLS traffic over loopback device? Can be switched off by attacker?)

One of the worst-case scenarios is that the attacker has access to a privileged sidecar container. A privileged container has all the capabilities of the host machine, so basically a root user inside a privileged container means that the attacker has root privileges on the host Node. Furthermore, the container lacks restrictions from Seccomp, AppArmor and Linux capabilities. The Listing 3 describes a Deployment with a privileged container with extremely insecure specification.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: privileged
  namespace: insecure-ns
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: privileged
  template:
    metadata:
      labels:
        app: privileged
    spec:
      hostPID: true # vector: nsenter works now for namespaces running outside the container
      hostNetwork: true # vector: sniff traffic with tcpdump, try to bypass NetworkPolicies
      hostIPC: true # vector: look for shared resources in /dev/shm and ipcs
      nodeName: k8s-control-plane-node # deploy workload on a control plane Node
      containers:
        - name: sidecar-container
          image: malicious-sidecar
          command: ["/bin/sleep", "3650d"]
          securityContext:
            privileged: true
          volumeMounts:
            - name: host-root
              mountPath: /host-root
      volumes:
        - name: host-root # vector: mount the host filesystem
          hostPath:
            path: /
```

Listing 3: Privileged container

The three first fields in the `spec` remove namespace isolations from the container and use the corresponding Node namespaces instead. With a privileged container and `hostPID: true`, the attacker can enter Node's namespace with `nsenter` (easy target PID is the **init** system running as 1), and execute commands on the host. The attacker can also see and enter other container's and processes running on the same Node. In a non-privileged container, the `hostPID` still allows Denial of Service (DoS) attacks by killing the processes on the Node.

Since the container mounts the host filesystem to a volume in `/host-root`, the attacker in a privileged container can trivially use `chroot` to execute commands as root user in the host Node context. Even without the explicit `hostPath` mount, the host machine's `/dev` is accessible from the privileged container, which means that the container can see the disk that contains hosts's filesystem and mount the disk. The attacker can use this technique to find any credentials stored on the host machine and

escalate the attack. Important credentials include for example `kubeconfig` files that store access tokens to the Kubernetes API server, ServiceAccount tokens that may have been mounted on any Pod on the host, SSH keys and hashed user passwords in `/etc/shadows`.

The biggest risks considering a non-privileged container with all the isolations intact relate to lateral movement and discovring of secrets. Kubernetes automatically mounts each container in a Pod with Service Account token, which can be used to attack the control plane if it is over-permissive. Since the cluster has no NetworkPolicies in place by default, any misconfigured control plane component, like Kubelet or API server with `anonymous-auth` set to true, could be exploited. In addition,the attacker can try to be breach or DoS any control plane component or workload in the cluster, or other connected services like those in a cloud hosted system. Furthermore, any unpatched vulnerability in the underlying kernel, container engine or Kubernetes can exploited.

The *malicious-sidecar* image is a simple, custom-made container image that includes some basic tools for penetration testing. The image also installs `kubectl` and sets environment variables so that the CLI tool works out-of-the-box with the default `minikube` installation. Its Dockerfile can be found in Appendix A. If the container has public internet access, the attacker could use a tool such as Docker Enumeration, Escalation of Privileges and Container Escapes (DEEPCE) [60] may be used to enumerate attack options even further.

4 Solution requirements

This chapter defines the security requirements of the solutions that are discussed in the following chapters. The chapter also introduces the development environment, which is used for testing and implementing the solutions.

4.1 Security requirements

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 the principle of least privilege; the sidecar inherits execution and networking privileges from the main application container. Based on the model, the solution should provide answers to these main questions:

1. How to ensure and enforce that no workloads that conflict with the 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 in which all mitigations are applied is introduced in Chapter 5. 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, Kubernetes does not provide any built-in solution for creating these firewalls. Few possible solutions for building the Zero Trust network inside the Pod are given in the Chapter 6.

4.2 Environment

The solution is tested and developed on a `Minikube` cluster running on a 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 (https://github.com/Arskah/k8s-sidecar-security). 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 6.

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 the loopback device on port `8125`, which is the default for StatsD.

For penetration testing purposes, another sidecar is introduced with common networking and Kubernetes command line tools. 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.

5 Hardening Pod security

This chapter provides a solution for hardening Pods against privilege escalation attacks. The solution enforces that deployed resources follow Kubernetes 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 a few oversights regarding sidecar containers in the controller.

5.1 Restricted Pod Security Standard

Since version 1.25, Kubernetes has shipped with the Pod Security Admission controller as a stable feature. The controller, as discussed in Chapter 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 at the best Pod hardening practices [13], so it will be the most optimal for our solution. The table 3 lists all fields affected by the standard.

Table 3: 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 group used by container.	Set both to non-zero to run as non-root.
windowsOptions.hostProcess	Runs Windows containers as privileged HostProcess.	false

```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 4: Namespace resource with `restricted` Security Standard

The Security Standard can be enforced trivially by adding `pod-security.kubernetes.io/enforce: restricted` as a label on a Kubernetes namespace resource, as shown in Listing 4.
JB: Since the work is on network security, it would be good to spend couple of words about the connection between the security standard and the networking.

5.2 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 straightforward: just add values to both `resources.limits.cpu` and `resources.limits.memory` fields for all of the containers, similarly to Listing 6. 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 [15]. For memory, the base unit is bytes, but it also supports quantity suffixes such as "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 an out-of-memory (OOM) error.

5.2.1 Manual service account mounting

By default, the Service Account tokens are mounted in the `/var/run/secrets/kubernetes.io/servic` 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 in 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.

```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 5: ServiceAccount with permissions for fetching Pod resources

Listing 5 shows how to create a Service Account token with the permission to 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 auto-mounting 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 Listing 6, 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 them, and the tokens cannot be used for privilege escalation from other containers of the Pod.

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app
  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 6: Two container Pod with resource limits and manually mounted ServiceAccount

5.2.2 Enforcing the fields

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

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

5.3 Solution

TODO: Summarize all the solutions into one.

6 Network Isolation

Implementing a zero trust network architecture in Kubernetes cluster is not trivial, since building network isolation between containers in the same Pod is not possible with common CNI plugins and Network Policies. The root cause for this is that the lowest level of networking abstraction in Kubernetes 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, passes through the same `eth0` NIC and shares a common source IP address. Since NetworkPolicies operate on L3/L4, they cannot distinguish whether traffic originates from the sidecar or from the main container. Although ingress traffic can be easily identified with the 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 intra-Pod communication, the traffic goes through the loopback device which is not managed by CNI plugins. JB: And by which component? I assume CRI (container runtime interface). Note for the future: Does it different significantly in different implementation of CRI? This means that NetworkPolicies do not have an effect on the loopback device. Furthermore, since all intra-Pod traffic has source and destination IP addresses of `127.0.0.1`, the packets have no clear identifiers that could be used for traffic filtering. This chapter introduces two general approaches as solutions for the issue.

The first solution enforces firewall rules inside the application Pod's network namespace, while the second approach creates network namespaces for each container by deploying them to their own Pods. While deploying containers in their own Pods is an obvious option for creating the isolation, the approach breaks the tight integration of sidecar containers to the application container. Most notably, the sidecars will have their own independent lifecycle and schedulation JB: scheduling, and the communication via the loopback device will be broken. In the scope of this thesis, we consider sidecars to be scheduled along the application container and be accessible via `127.0.0.1`. Thus, re-introducing these characteristics to the containers is also part of the solution.

6.1 Applying firewall rules inside Pod network namespace

Figure 5 illustrates the network architecture with all required network rule enforcement points. The solution needs to enforce network rules in four scenarios: i) external components and Pod's `eth0`, ii) main container and `lo`, iii) sidecar container and `lo`, and sidecar container and `eth0`. The ingress and egress rules in the first scenario are more easily enforced with `NetworkPolicies`, as shown in Listing 7. The policies allow cluster and external traffic to the main application port `8888`, while denying all other traffic into and out of the Pod.

```
1 apiVersion: networking.k8s.io/v1
2 kind: NetworkPolicy
3 metadata:
```

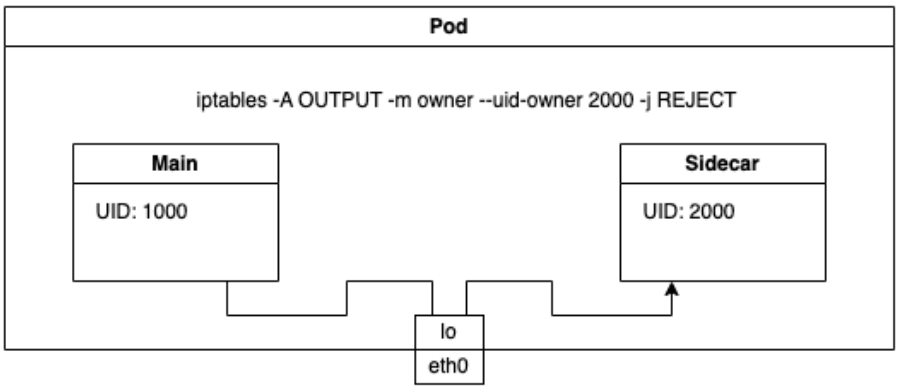


Figure 5: Single network namespace architecture

```
  name: deny-all
  namespace: app
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: web-allow-external
  namespace: app
spec:
  podSelector:
    matchLabels:
      app: node-app
  ingress:
  - ports:
    - port: 8888
```

Listing 7: Network Policy handling traffic between Pod and external components

The other scenarios cannot be enforced with NetworkPolicies, since the rules depend on individual containers instead of Pods. The rules need to be applied inside the Pod's network namespace with tools such as IPTables.

6.1.1 IPTables

Usually, IPTables rules are applied to packets by their IP addresses and ports. However, modern versions of IPTables ship with an owner module, which supports rules based

on application's user, group, and session identifiers. For example, `iptables -A OUTPUT -m owner -uid-owner 1000 -j REJECT` would reject all egress packets from containers with the user id. While the session identifier is assigned at runtime by the kernel, containers' user and group ids can be overridden with Pod definition's `runAsUser` and `runAsGroup` fields, respectively. Thus, any ingress or egress traffic of a container with unique user identifier can be filtered with IPTables.

Listing 8 shows IPTables rules that can enforce network rules in other scenarios. As ingress rules, the main application port is opened on all network devices, while the sidecar port is only allowed on the loopback device. For egress traffic, the main application is given access to both containers' ports, while the sidecar can only access its own port. All other traffic is rejected by default.

```
#!/bin/bash
APP_UID=1000
APP_PORT=8888
SIDECAR_UID=2000
SIDECAR_PORT=8125

# Ingress rules
iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j
    ACCEPT
iptables -A INPUT -p tcp --dport $APP_PORT -j ACCEPT
iptables -A INPUT -p udp --dport $SIDECAR_PORT -j ACCEPT
iptables -A INPUT -j DROP

# Egress rules
iptables -A OUTPUT -m conntrack --ctstate RELATED,ESTABLISHED -j
    ACCEPT

## Main app rules
iptables -A OUTPUT -o lo -m owner --uid-owner $APP_UID -p tcp --
    dport $APP_PORT -j ACCEPT
iptables -A OUTPUT -o lo -m owner --uid-owner $APP_UID -p udp --
    dport $SIDECAR_PORT -j ACCEPT

## Sidecar rules
iptables -A OUTPUT -o lo -m owner --uid-owner $SIDECAR_UID -p udp
    --dport $SIDECAR_PORT -j ACCEPT

iptables -A OUTPUT -j REJECT
```

Listing 8: IPTables rules for the Pod

The approach is similar to how Istio redirects all Pod traffic through the Envoy sidecar and the service mesh. Istio adds IPTables rules that reroute all traffic, except that which originates from user id 1337, to the Envoy proxy [50]. The user id itself is reserved for the proxy, and it is used to avoid re-routing the proxy traffic, which would cause an infinite loop.

eBPF programs were also considered as an option for IPTables. Since XDP programs support only ingress rules, they are not a valid option for solving the problem. However, a TC program that could distinguish sidecar container traffic from

others could be a possible option. TC programs manipulate packets by modifying `struct __sk_buff` in the user space. The complete C structure is listed as Listing 9.

```
/* user accessible mirror of in-kernel sk_buff.
 * new fields can only be added to the end of this structure
 */
struct __sk_buff {
  __u32 len;
  __u32 pkt_type;
  __u32 mark;
  __u32 queue_mapping;
  __u32 protocol;
  __u32 vlan_present;
  __u32 vlan_tci;
  __u32 vlan_proto;
  __u32 priority;
  __u32 ingress_ifindex;
  __u32 ifindex;
  __u32 tc_index;
  __u32 cb[5];
  __u32 hash;
  __u32 tc_classid;
  __u32 data;
  __u32 data_end;
  __u32 napi_id;

  /* Accessed by BPF_PROG_TYPE_sk_skb types from here to ... */
  __u32 family;
  __u32 remote_ip4;    /* Stored in network byte order */
  __u32 local_ip4;     /* Stored in network byte order */
  __u32 remote_ip6[4]; /* Stored in network byte order */
  __u32 local_ip6[4];  /* Stored in network byte order */
  __u32 remote_port;   /* Stored in network byte order */
  __u32 local_port;    /* stored in host byte order */
  /* ... here. */

  __u32 data_meta;
  __bpf_md_ptr(struct bpf_flow_keys *, flow_keys);
  __u64 tstamp;
  __u32 wire_len;
  __u32 gso_segs;
  __bpf_md_ptr(struct bpf_sock *, sk);
  __u32 gso_size;
  __u8  tstamp_type;
  __u32 :24;    /* Padding, future use. */
  __u64 hwtstamp;
};
```

Listing 9: `skbuff` struct in Linux v6.3 [34]

As can be seen in the example, the structure has no identifiers that are unique for each container. A notable attribute in the structure is the `mark` field, which can be used as an identifier for the packet in the kernel. It does not propagate with the packet, which means that marking can only be used for traffic filtering

before the packet leaves the network stack. If combined with `iptables` rule that uniquely marks the packets egressing from the sidecar, for example by setting the mark as 2 `iptables -t mangle -A PREROUTING -m owner -uid-owner 1000 -j MARK -set-mark 2`, a TC eBPF program could be used for filtering the traffic. However, using eBPF brings an extra layer of complexity in comparison to simply using IPtables for traffic filtering. Furthermore, eBPF programs do not provide any performance gains for egressing traffic, since the TC hooks execute after the packet has been processed by IPTables.

6.1.2 Deployment

The commands for creating the IPTables rules could be applied to the Pod with the initContainer or postStart lifecycle handler. However, the commands require root user permissions with NET_ADMIN and NET_RAW capabilities. Using these methods would require changing Pod's security admission level to privileged. Thus, it is more optimal to apply the rules outside the Pod's context; for example, by using a DaemonSet, similarly to how many CNI plugins configure the Pod network.

6.2 Own network namespace for sidecar

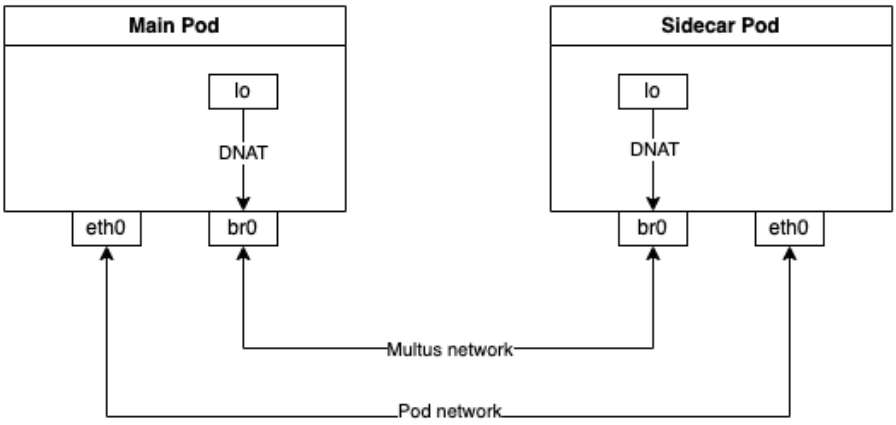


Figure 6: Multiple network namespaces architecture

Figure 6 illustrates the network architecture for the solution in which containers are deployed in different Pods. The idea behind this approach is to use Network Policies for filtering the traffic between the containers. Since containers are not technically sidecars anymore, the solution must recreate communication via the `localhost` address.

Although it is possible to use the default network created by the CNI plugin for sidecar traffic, the proposed solution creates a new isolated network for traffic with Multus. The isolation allows the use of separate IP address pool, which comes in handy when re-creating the communication via localhost.

6.2.1 Creating isolated network for sidecar traffic

It is completely possible to use the default Pod network created by the CNI plugin for traffic between sidecars. With this setup, the network rules are simply Network Policies on top of the policy that implicitly denies all other traffic.

It is also possible to build new network interfaces for sidecar traffic with Multus.

```
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"
      }
    }'
```

Listing 10: NetworkAttachmentDefinition for creating MacVLAN bridge between the Pods

The setup requires creating `NetworkAttachmentDefinition` CRD for the network. Each Pod that should be part of the network needs a `k8s.v1.cni.cncf.io/networks` annotation, as seen in Listing 11.

```
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:
      containers:
        - name: node-app
          image: node-app
---
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:
      containers:
        - name: statsd
          image: hypnza/statsd_dumpmessages
```

Listing 11: Deployment with Pod connected to the bridge

The common NetworkPolicies are not applied to the Multus network. However, the Multus developers have introduced another CRD, `MultiNetworkPolicy` [40], which mimics NetworkPolicies but only works for networks created by Multus. The resource is identical to NetworkPolicies, as seen in the Listing 12.

```
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:
  podSelector:
    matchLabels:
      app: node-app-statsd
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: node-app
      ports:
        - protocol: TCP
          port: 8125
```

Listing 12: Zero trust MultiNetworkPolicies for br0 network interface

However, the Multus CNI does not enforce the `MultiNetworkPolicy` rules in any way. The implementation is left for other DaemonSets such as `multi-networkpolicy-iptables` [41] and `multi-networkpolicy-tc` [42], both of which are created by the Multus team. As the names suggest, the DaemonSets implement the enforcement with IPTables and eBPF TC programs, respectively. The `multi-networkpolicy-iptables` is used for the solution, since `iptables` implements all other network rules for the cluster.

With these configurations and dependencies, the containers should have a new network and reach each other via the bridge interface. As the next step, the solution must find a way to redirect the traffic destined for the localhost to the new network. The script in Listing 13 shows how to redirect all egressing traffic with IPTables from the source of `127.0.0.1:8125` to another network interface with an external IP address, like `192.168.1.202:8125`. By default, the re-routing of local traffic outside the local machine is denied by the kernel. To overrule this default behavior, the script sets a kernel flag with net.ipv4.conf.all.route_localnet=1.

```
#!/bin/bash
sysctl -w net.ipv4.conf.all.route_localnet=1
```

```
3 iptables -t nat -A OUTPUT -m addrtype --src-type LOCAL --dst-type
    LOCAL -p udp --dport 8125 -j DNAT --to-destination
    192.168.1.202:8125
4 iptables -t nat -A POSTROUTING -m addrtype --src-type LOCAL --dst-
    type UNICAST -j MASQUERADE
```

Listing 13: Script for mapping localhost to br0 interface

6.2.2 Mutating sidecars to multiple Pods?

MutatingAdmissionWebhook?

7 Solution Evaluation

8 Discussion

9 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. *Custom Resources*. 2023. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/ (visited on 07/16/2023).

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

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

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

[11] Kubernetes Authors. *Operator pattern*. 2023. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (visited on 07/16/2023).

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

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

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

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

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

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

[18] 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.

[19] Thanh Bui. "Analysis of docker security". In: *arXiv preprint arXiv:1501.02967* (2015).

[20] 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.

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

[22] 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.

[23] The MITRE Corporation. *Containers Matrix*. 2023. URL: https://attack.mitre.org/matrices/enterprise/containers/ (visited on 07/27/2023).

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

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

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

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

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

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

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

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

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

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

[34] Linux developers. *bpf.h in Linux source code*. 2023. URL: https://github.com/torvalds/linux/blob/v6.3/include/uapi/linux/bpf.h (visited on 06/23/2023).

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

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

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

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

[39] 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).

[40] Kubernetes Network Plumbing Working Group. *multi-networkpolicy CRD*. 2023. URL: https://github.com/k8snetworkplumbingwg/multi-networkpolicy (visited on 07/03/2023).

[41] Kubernetes Network Plumbing Working Group. *multi-networkpolicy-iptables*. 2023. URL: https://github.com/k8snetworkplumbingwg/multi-networkpolicy-iptables (visited on 07/03/2023).

[42] Kubernetes Network Plumbing Working Group. *multi-networkpolicy-tc*. 2023. URL: https://github.com/k8snetworkplumbingwg/multi-networkpolicy-tc (visited on 07/03/2023).

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

[44] 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).

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

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

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

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

[49] 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.

[50] Istio. *Install Istio with the Istio CNI plugin*. 2023. URL: https://istio.io/v1.17/docs/setup/additional-setup/cni/ (visited on 06/14/2023).

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

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

[53] 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.

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

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

[56] 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.

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

[58] Francesco Minna et al. "Understanding the security implications of kubernetes networking". In: *IEEE Security & Privacy* 19.5 (2021), pp. 46–56.

[59] 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.

[60] Matthew Rollings. *Deepce: Container enumeration and exploitation script*. 2023. URL: https://stealthcopter.github.io/deepce/ (visited on 07/28/2023).

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

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

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

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

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

[66] 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://packages.cloud.google.com/apt/doc/apt-key.gpg
RUN echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main' | tee /etc/apt/sources.list.d/kubernetes.list

# 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/kubeletctl_linux_amd64 && chmod a+x ./kubeletctl_linux_amd64 && mv ./kubeletctl_linux_amd64 /usr/local/bin/kubeletctl

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' >> root/.bashrc
# 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 14: 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 15: 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 16: `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 17: Dockerfile for the server

C Webserver and StatsD deployed with Multus networking

```
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
            resources:
              limits:
                cpu: "1"
                memory: 1024Mi
---
apiVersion: v1
kind: Service
metadata:
  name: node-app
  namespace: app
spec:
  ports:
  - port: 8888
    protocol: TCP
    nodePort: 30001
  selector:
    app: node-app
  type: NodePort
```

Listing 18: Server's K8s deployment resource

```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:
```

```
45                 type: RuntimeDefault
46               capabilities:
47                 drop:
48                   - ALL
49             resources:
50               limits:
51                 cpu: "1"
52                 memory: 256Mi
```

Listing 19: StatsD container's K8s Deployment

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: app
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
---
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: 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:
  podSelector:
    matchLabels:
      app: node-app-statsd
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: node-app
      ports:
        - protocol: TCP
          port: 8125
```

Listing 20: Network interfaces and policies