A” **Aalto University**
School of Science

Master's Programme in Computer, Communication and Information Sciences

Security Risks for Sidecar Containers in Kubernetes

Aarni Halinen

Master's Thesis
2023

A'' **Aalto University**
School of Science

Author Aarni Halinen	
Title Security Risks for Sidecar Containers in Kubernetes	
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 29.12.2023 **Number of pages** 62+11 **Language** English

Abstract

The widespread adoption of containers has been a driving force behind the shift from monolithic applications to microservices. The modular nature of microservice architecture has allowed the development of distributed system design patterns such as the sidecar pattern, in which peripheral tasks are separated from the application containers as their own distinct containers. The sidecar pattern is often used to implement observability, logging, and improving security with service meshes.

This thesis examines the security risks associated with sidecar containers in Kubernetes container orchestration. It focuses on Kubernetes networking and Zero Trust security architecture to identify potential vulnerabilities that adversaries could exploit. The research reveals that Kubernetes only offers built-in security measures on the Pod-level, which can leave sidecar containers exposed to attacks.

The thesis introduces two proof-of-concept solutions aimed at establishing network isolation between application and sidecar containers. The first solution utilizes IPTables to enforce isolation, while the second employs Multus to mimic shared network namespace between containers in distinct Pods.

Despite the demonstrated solutions for network isolation, the complexities of implementation and the need for specialized infrastructure pose challenges that may outweigh the benefits of such architectural modifications. Due to the challenges, the thesis explores the merits of sidecarless service mesh architectures such as Cilium and Istio Ambient Mesh, as an example of avoiding the risks with sidecars altogether.

Ultimately, while the thesis presents viable proof-of-concept solutions for network isolation between application and sidecar containers, it advocates for a cautious approach, emphasizing the preference for established and well-tested components over custom-made implementations that need to be actively maintained.

Keywords Kubernetes, Container, Sidecar pattern, Network Security

Tiivistelmä

Konttiteknologian yleistyminen on ollut merkittävä tekijä siirryttäessä monoliittisista sovelluksista mikropalveluihin. Mikropalveluarkkitehtuurin modulaarisuus on mahdollistanut hajautettujen järjestelmäsuunnittelumallien, kuten sivuvaunumallin, kehittymisen. Sivuvaunumallissa sovelluksesta erilliset sivutehtävät kuten havainnointi, lokien kirjoittaminen tai palveluverkkojen välityspalvelimet voidaan toteuttaa omina erillisinä kontteinaan pääsovelluksen rinnalla.

Tämä diplomityö tutkii turvallisuusriskejä, jotka liittyvät sivuvaunukontteihin Kubernetes-orkestraatiossa. Työ keskittyy Kuberneteksen verkkototeutukseen ja nollaluottamuksen tietoturva-arkkitehtuuriin ja tutkii sivuvaunukontteihin liittyviä tietoturvariskejä, joita hyökkääjät voisivat hyödyntää. Kubernetes tarjoaa valmiita tietoturvaominaisuuksia vain Pod-abstraktiotasolla, mikä altistaa sivuvaunukontteja hyökkäyksille.

Työssä esitellään kaksi alustavaa ratkaisua, joiden avulla voidaan eristää sovellus- ja sivuvaunukonttien verkot toisistaan. Ensimmäinen ratkaisu perustuu IPTables-sääntöihin, kun taas toinen luo Multus-konttiverkkoliitännäisen avulla sivuvaunumallia imitoivan verkon kahden eristetyn kontin välille.

Molemmat toteutukset ratkaisevat ongelman, mutta niiden monimutkaisuus ja tarve erikoistuneelle intrastruktuurille voi merkittävästi hankaloittaa ratkaisujen käyttöönottoa ja hyödyllisyyttä tuotantoympäristöissä. Kohdattujen haasteiden vuoksi työ tutkii myös sivuvaunuttomia palveluverkkoja, kuten Cilium ja Istio Ambient Mesh, keinona välttää sivuvaunuihin liittyvät turvallisuusriskit.

Vaikka diplomityössä esitetäänkin ratkaisut sivuvaunukonttien verkkoeristykseen, on välttämätöntä korostaa vakiintuneiden ja hyvin testattujen komponenttien suosimista räätälöityjen ja aktiivista ylläpitoa vaativien toteutusten sijaan.

Avainsanat Kubernetes, konttiteknologia, sivuvaunumalli, verkkoturvallisuus

Preface

I want to thank my advisors, Jacopo Bufalino and José Luis Martin Navarro, for their valuable advice and guidance for this thesis. I also want to thank my supervisor Mario Di Francesco for all his professional comments and support during the thesis process and earlier in my master's studies.

I also would like to thank all my friends, Radiodiodi and Sähköinsinöörikilta, for all the memorable moments during my (many) years of studies. Last but not least, I want to thank my family for their support and encouragement during my studies and this thesis.

Helsinki, 29 December 2023

Aarni Halinen

Contents

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
mTLS	Mutual Transport Layer Security
NAT	Network Address Translation
NIC	Network Interface Controller
OOM	Out-of-memory
OpenVZ	Open Virtuozzo
OS	Operating System
OSI	Open Systems Interconnection
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 moved away from monolithic software applications and towards microservices. The microservice architecture divides an application into a set of smaller, independent services that each handle a part of the business logic [46]. Each service runs its own process, and the application data is sent between components using lightweight communication protocols such as the Hypertext Transfer Protocol (HTTP) and Google Remote Procedure Call (gRPC). This approach increases software agility, as each service becomes an independent unit of development, deployment, operations, versioning, and scaling [62]. This modularity is often associated with benefits such as faster delivery and better scalability.

The increasing use of containers has been a driving force behind the shift to microservices. Containers are a lightweight virtualization technique where an application and all its dependencies are bundled into a single deployable unit running on the host machine kernel [22]. In a container-based microservice architecture, each service is bundled in its own container and deployed independently from the other services. Popular container runtimes include *Docker*, *containerd*, and *CRI-O*.

For more complex systems with multiple containers, scaling needs, and fault tolerance, a container orchestrator can be used. Container orchestration tools automate the deployment, management, scaling, and networking of container workloads and handle tasks such as resource allocation and health monitoring. The container orchestrators operate on top of a group of host machines, called a cluster, which serves as a resource pool for all container workloads. Kubernetes is the most widely used orchestrator tool.

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 [23]. 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 the sidecar pattern are similar to microservices, allowing better resource allocation, re-use of components in other services, and providing 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.

Kubernetes offers limited security features at the container level. Although containers have their own distinct security context definitions, all security-related policies and features are defined for Pods or Namespaces. This means that sidecars, even if not required, inherit any capability and privileges from the main application. Thus, any privileged workload, even in another container in the Pod, risks privilege escalation from the sidecar. Additionally, Kubernetes' firewalling solution, NetworkPolicies, is applied to the entire Pod instead of individual containers. This means that there are no built-in mechanisms in Kubernetes for protecting and isolating containers running in the same Pod from each other, which goes against the principle of least privilege. Consequently, any exploitable security issue in a sidecar container could allow the attacker to move laterally to other containers in the Pod and escalate privileges. This makes the sidecar an ideal starting point for attacks.

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 the container-level, we need to find some other ways to implement these paradigms inside the Pods. This thesis proposes solutions for restricting the capabilities of sidecars while minimally affecting the main container, thus improving security by extending the paradigms within the Pod.

1.2   Thesis outline

The following Chapter 2 gives background about containers and Kubernetes, and discusses about Kubernetes networking and container network interface plugins. Chapter 3 analyzes security threats in a Kubernetes cluster with sidecars based on existing threat models and also demonstrates potential attack scenarios. Chapter 4 defines the requirements of the solution and introduces the development environment used for the solutions. Chapters 5 and 6 propose solutions to mitigate security threats by hardening Pods and setting up network isolation, respectively. Chapter 7 evaluates the pros and cons of the solutions. Finally, Chapter 8 discusses future research and concludes the thesis.

2 Background

2.1 Zero trust architecture

Conventional network security has historically focused on perimeter defense [63]. 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 [63, 73]. 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 that 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 access. This achieves a multi-layered security boundary, 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.

Zero trust networking is based on the concept that the network, even if it is internal and behind a firewall, should never be trusted. Instead of authenticating and authorizing only at the boundary, trust is verified at each component of the network. This implies that each service should communicate securely in the network using protocols such as Mutual TLS (mTLS), and rely on more precise identifiers than just IP addresses. Furthermore, access to a resource should be as limited as possible, and the policy can be further refined to restrict access based on, for example, HTTP routes on L7. The ultimate aim in all cases is to achieve a policy of least privilege.

The complexity of managing the trust boundary in each component can be overwhelming. Fortunately, Kubernetes can be extended with service meshes such as *Istio*. Service meshes provide a dedicated infrastructure layer in the application that allows the addition of features such as observability, traffic management, and security without having to modify the application code [36]. Istio consists of two components: the data plane, which is composed of Envoy proxies that are deployed along with each service in the cluster, and the control plane. The proxies intercept all network traffic of the parent service and implement features such as mTLS and traffic management, while also sending logs and other telemetry data to the control plane.

When the `istio-injection=enabled` label is applied to a Namespace, any new Pods created in that Namespace will automatically have a sidecar injected into them.

2.2 Containerization and Docker

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

Figure 1: Virtualization models [25]

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 [67]. 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 [25]. 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 [44] is a hugely popular LXC-based container runtime and provides an easy-to-use API and tooling for creating and managing containers. Docker also provides containerization for Windows and Mac operating systems 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 [65]. Namespaces [41] is 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) which isolates POSIX message queues, etc., iii) *Network* which isolates network devices, stack ports, etc., iv) *Mount* for file system isolation, v) *Process ID* (PID), vi) *Time*, vii) *User* for isolating user and group identifiers, and viii) *UTS* which isolates hostnames and NIS domain names. For example, the *network* namespace provides each container with its own loopback device and even IPTables rules. In another example, the *mount* namespace ensures that the 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 [65]. 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 using cgroups.

Since all containers and the host machine run on the same kernel, any container that manages to break out 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 [65]. These include Discretionary Access Control (DAC) mechanisms like Capability [39] and Secure computing mode (seccomp) [42], 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 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 an 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 [45]. The Seccomp mechanism constrains the system calls that 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 [43] 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 [65].

2.2.2 Docker

Docker is an open-source container technology written in Go and launched in 2013 [3]. The platform consists of the Docker Engine, a container runtime and packaging tool, and Docker image registries such as the Docker Hub public image repository and the Docker desktop application [2]. The engine functions as a client-server RESTful HTTP API application, with the Docker daemon acting as a server, managing and executing all the containers. As illustrated in Figure 2, the daemon serves as a high-level container runtime and uses *runC* and *containerd* to create and run the

Figure 2: Docker architecture [27]

containers in a system.

The Docker client is a command-line tool that provides a user interface to command 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 [22]. 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.

2.3 Kubernetes

While container runtimes are used to create and destroy containers, container orchestrators are used as a system to automate the deployment, scaling, and management of containerized applications. Kubernetes (K8s) [10] is an open-source container orchestrator released in 2014. It is hosted by the Cloud Native Computing Foundation (CNCF), but its roots are at Google, where it was developed as an alternative to

14

Google's proprietary Borg and Omega orchestrators [24]. A Kubernetes cluster consists of a set of servers, called **Nodes**, on which application containers are scheduled by the system. This automation provides resilience and efficient resource utilization for workloads in the cluster: if a container or Node fails, the system attempts to restart and reschedule containers to maintain the desired cluster state.

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 [16]. Containers in a Pod are always co-located and co-scheduled and run in a shared context, i.e. a set of Linux namespaces. Network, UTS, and IPC namespaces are shared by default, and 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 [16]. 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 Chapter 2.4.1.

All Pods in the cluster share the same subnet and can access each other via IP address. However, connecting to a Pod with an 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 the following 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 [18]. 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.

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 must 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 multitenancy setups. Pod Security Standards [15], 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 constructs 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* [13]. 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 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 [11]. 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 the 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

Figure 3: Kubernetes cluster architecture [11]

(CLI), 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 [11]. 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 [11]. 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 toward the desired state. All 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 [11]. It receives a set of Pod specifications 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 the network rules on Nodes. Part of the Service objects' networking is implemented by **kube-proxy**; the proxy writes IPTables rules that route traffic [31].

17

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** resources for building custom 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 the 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 allowed in the namespace to which the Pod is assigned [5]. The controller is enabled by default and can be taken into use simply 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` [15].

The available actions are i) `enforce`, which will reject the Pod on a violation, ii) `audit`, which triggers an event about the violation in the audit log, and iii) `warn`, which triggers a user-facing warning about the violation [14]. 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 a 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 [23]. 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 logs can be shared to the sidecar with file mounts. Another example of this pattern is the Istio service mesh [36] and its Envoy proxy sidecar, which routes all traffic through the Istio control plane for management, observability, and security operations.

```yaml
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, called 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 to using separate containers [23]. 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 in a way that they can be reused with other application containers. From a testing point of view, the componentized system could improve testing if the smaller units could 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 [71]. All agents on a host should also be able to communicate with Pods on the same host. Kubernetes does not provide a built-in solution for this model; instead, it entrusts the implementation to the 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 [19] is a networking specification, which has become the de-facto industry standard for container networking and is backed by CNCF [71]. The specification was first developed for the container runtime *rkt* [59], but it is now supported by almost 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 [19]. The network configuration is defined as a JSON file that 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 the runtime mutates the 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 in a 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 containers [58]. 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 to the entire 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. 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 an 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 `PodCIDRs` 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 capability 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 NetworkPolicies 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
1  apiVersion: networking.k8s.io/v1
2  kind: NetworkPolicy
3  metadata:
4    name: deny-all
5    namespace: frontend
6  spec:
7    podSelector: {}
8    policyTypes:
9      - Ingress
10     - Egress
11 ---
12 apiVersion: networking.k8s.io/v1
13 kind: NetworkPolicy
14 metadata:
```

```
15    name: webapp-ingress
16    namespace: frontend
17  spec:
18    podSelector:
19      matchLabels:
20        app: webapp
21    policyTypes:
22      - Ingress
23    ingress:
24      - ports:
25          - protocol: TCP
26            port: 80
27  ---
28  apiVersion: networking.k8s.io/v1
29  kind: NetworkPolicy
30  metadata:
31    name: webapp-egress
32    namespace: frontend
33  spec:
34    podSelector:
35      matchLabels:
36        app: webapp
37    policyTypes:
38      - Egress
39    egress:
40      - to:
41          - namespaceSelector:
42              matchExpressions:
43                - key: namespace
44                  operator: In
45                  values: ["backend"]
```

Listing 2: Example policies 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 traffic [21] to the target Pod. Each rule includes `to` or `from` field for selecting the 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, 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 above specifications, 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 [75] is an open-source CNI plugin with a 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 [56]. 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. To route 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 sets up the 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 consisting of controller loops for Namespace, NetworkPolicy, Node, Pod, and ServiceAccount Kubernetes objects. The Calico project also introduces its own CLI tool, called `calicoctl` [77], to manage Calico's custom resources. The tool provides additional 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 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., matching on ServiceAccounts) [76]. The policy can also match other Calico CRDs such as **HostEndpoints** and **NetworkSets**, which allows the implementation of rules on host interfaces and non-Kubernetes resources. If Calico is installed along the Istio service mesh, the Calico NetworkPolicy can enforce L7 (e.g. HTTP methods and URL paths) policies on the Envoy proxy. For policies that are not tied to a Kubernetes namespace, Calico provides a **GlobalNetworkPolicy** CRD.

2.4.3 Cilium

Cilium [28] 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 [57] 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 [21]. 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 [28].

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*) [29]. 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 remotely 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 [30]. If L7 policies exist, traffic is directed to the Envoy instance bundled into the agent Pod which filters traffic. Unlike in layers 3 and 4, policy violation does not result in a 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 a **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 native *kube-proxy* replacement, encryption for Cilium-managed traffic, and service mesh, among others. By default, *kube-proxy* uses IPTables to route Service traffic [31]. With the `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 [33]. 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 [48]. 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.

2.4.4 Multus

Traditionally, CNI plugins provide only a single network interface for a Pod, apart from the loopback device. Multus [54] 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 [54]. 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 [55]. 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 a 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 annotation 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 [66]. 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 a 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. Instructions are executed in a register-based pseudo machine. Since network monitors are often interested only in a 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 the *libpcap* library, which is used by a network monitoring tool called `tcpdump`.

Later in the 2010s, the Linux community realized that BPF and its ability to instrument the kernel could also benefit other areas beyond packet filtering [78]. 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 the technology. Similarly to the kernel community, the term BPF always refers to eBPF in this thesis.

25

The eBPF programs are compiled to bytecode and loaded into the kernel with the `bpf()` system call [68]. 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 the `bpf_asm` utility to convert the 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) [68]. XDP programs are attached to an NIC and can handle only incoming packets [60]. 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 [32]. 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 [78]. 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 the `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 [78]. 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.

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 [74].

For Kubernetes clusters, there exists an extensive *Threat Matrix for Kubernetes* created by Microsoft [69]. The matrix is adapted from *MITRE adversarial tactics, techniques, and common knowledge (ATT&CK)* Framework's container matrix [26] 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 [70]. 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 in this thesis 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 into 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 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 actors can break out from the container and escalate the attack on the 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	If allowed, malicious actors can try to tamper with the application or write executables to a disk.	Do not allow unnecessary mounts. Use `readOnlyRootFilesystem: true` whenever possible.
Permissive RBAC on ServiceAccounts	Containers in a Pod share ServiceAccounts[1]. By default, Pods automatically mount the ServiceAccount to all containers.	Disable automatic mounting of ServiceAccounts[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.	**No built-in solution available!**
Sidecar sniffs the main application traffic	Pod-to-Pod traffic in a cluster is unencrypted by default. A privileged sidecar can sniff and hijack traffic.	CNIs and service meshes may provide encryption to inter-Pod traffic with, for example, IPSec, Wireguard, and mutual TLS. These are often implemented with sidecars, which is not feasible in this scenario. For intra-Pod traffic, containers would need to implement this as part of the containers.

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 specifications.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: privileged
5    namespace: insecure-ns
6  spec:
7    replicas: 1
8    selector:
```

```
 9    matchLabels:
10      app: privileged
11  template:
12    metadata:
13      labels:
14        app: privileged
15    spec:
16      hostPID: true # vector: nsenter works now for namespaces
    running outside the container
17      hostNetwork: true # vector: sniff traffic with tcpdump,
    try to bypass NetworkPolicies
18      hostIPC: true # vector: look for shared resources in /dev/
    shm and ipcs
19      nodeName: k8s-control-plane-node # deploy workload on a
    control plane Node
20      containers:
21        - name: sidecar-container
22          image: malicious-sidecar
23          command: ["/bin/sleep", "3650d"]
24          securityContext:
25            privileged: true
26          volumeMounts:
27            - name: host-root
28              mountPath: /host-root
29      volumes:
30        - name: host-root # vector: mount the host filesystem
31          hostPath:
32            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 containers and processes running on the same Node. In a non-privileged container, `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 the 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 the host'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 the discovery of secrets. Kubernetes automatically mounts each container in a Pod with a ServiceAccount 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, such as Kubelet or API server with `anonymous-auth` set to true, could be exploited. In addition, the attacker can try to 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 runtime, 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) [72] 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. This chapter also introduces the development environment which is used to test and implement the solutions.

4.1 Security requirements

Threat modeling identified two main categories of issues: permissive workload configurations and networking-related issues. Essentially, all 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 workloads that conflict with the principle of least privilege are not deployed to the cluster?

2. How to enforce Zero trust network that allows traffic filtering on the container-level and limits communication on both loopback and Pod network interface devices?

Regarding the first question, the existing mitigations were already listed in Table 1 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. A few possible solutions for building the Zero trust network inside the Pod are given in Chapter 6.

4.2 Environment

The solution is tested and developed on a Minikube v1.32.0 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 were tested and considered as CNI plugins. Neither CNI had implications on the behavior of networking solutions. However, Cilium was selected on the basis that it does not use IPTables, and thus offered a less noisy debugging experience compared to Calico.

A simple Node.js webserver with a StatsD metrics aggregation sidecar is used as the application workload. The source code for the webserver can be found in Appendix B, while a StatsD debugging container called `hypnza/statsd_dumpmessages` is downloaded from Dockerhub [61]. The webserver serves "Hello world" on port `8888` and

sends response times and other metrics as UDP messages to the StatsD sidecar on default port `localhost:8125`. The sidecar prints incoming stats verbosely to stdout.

For penetration testing purposes, the *malicious-sidecar* container is used. 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.

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 additional 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 [15], so it will be the most optimal for our solution. The Security Standard can be easily enforced by adding `pod-security .kubernetes.io/enforce: restricted` as a label on a Kubernetes namespace resource, as shown in Listing 4.

```
1 apiVersion: v1
2 kind: Namespace
3 metadata:
4   name: foo
5   labels:
6     name: foo
7     pod-security.kubernetes.io/enforce: restricted
8     pod-security.kubernetes.io/audit: restricted
9     pod-security.kubernetes.io/warn: restricted
10    pod-security.kubernetes.io/enforce-version: latest
11    pod-security.kubernetes.io/audit-version: latest
12    pod-security.kubernetes.io/warn-version: latest
```

Listing 4: Namespace resource with `restricted` Security Standard

Table 3 illustrates the fields that the restricted security standard affects. From the table, it can be noted that the standard already blocks 2 out of the 4 identified threats: the deployment of privileged containers and host directory mounts. However, it should be noted that the standard restricts the networking solution discussed in the following chapter. Since the security standard only allows for the NET_BIND_SERVICE capability, any Pod attempting to modify the network stack will not work. Therefore, network modifications must be implemented outside of the Pod's context, which is discussed in greater detail in Section 6.1.2.

Table 3: Pod fields enforced by `restricted` Security Standard

Field name	Usage	Allowed values
`hostPID`, `hostIPC`, `hostNetwork`	Controls whether the container uses the host's PID, IPC, and network namespace.	`false`
`privileged`	Controls whether Pod can run privileged containers.	`false`
`capabilities` `.add` / `drop`	Defines Linux capabilities for the container.	Add `NET_BIND_SERVICE` allowed, drop `ALL` required
`volumes[*]`	All volume types are not allowed. For example, hostPath, which maps host directories, is not allowed.	`configMap`, `csi`, `downwardAPI`, `emptyDir`, `ephemeral`, `persistentVolumeClaim`, `projected`, `secret`
`hostPort`	Expose the container via the host's network port.	Must not be defined
AppArmor annotation[1]	Sets the AppArmor profile used by containers.	`runtime/default`, `localhost/*`
`seLinuxOptions`	Sets the SELinux context of the container.	Set if supported by the environment.
`procMount`	The default /proc masks are set up to reduce the attack surface.	`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.	All disallowed except for an allowed "safe" subset [2]
`allowPrivilegeEscalation`	Controls whether the process can gain more privileges than the parent.	`false`
`runAsNonRoot`	Controls whether the container can run as the root user.	`true`
`runAsUser`, `runAsGroup`	Controls the user and group used by the container.	Set both to non-zero to run as non-root.
`windowsOptions` `.hostProcess`	Runs Windows containers as privileged HostProcess.	`false`

---

[1]`container.apparmor.security.beta.kubernetes.io/*`

[2]`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`

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 ServiceAccounts 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, similar to the Listing 6. The CPU usage is measured in CPU units and can also be expressed in millicpus, i.e., both an integer value of 1 and "1000m" are equivalent to 1 physical or virtual core [17]. For memory, the base unit is bytes, but it also supports quantity suffixes such as "M", "Mi", and "Gi" for megabytes, mebibytes, and gigabytes, respectively. The resource limits are registered to the container's cgroups by the Kubelet. Limits are hard, which means that if a container exceeds its CPU limit, 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 ServiceAccount mounting

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

```
1  apiVersion: rbac.authorization.k8s.io/v1
2  kind: Role
3  metadata:
4    namespace: foo
5    name: foo-service-account-role
6  rules:
7  - apiGroups: [""] # "" indicates the core API group
8    resources: ["pods"]
9    verbs: ["get", "watch", "list"]
10 - apiGroups: [""] # "" indicates the core API group
11   resources: ["pods"]
12   verbs: ["get", "watch", "list"]
13 ---
14 apiVersion: v1
15 kind: ServiceAccount
16 metadata:
17   name: foo-service-account
18   namespace: foo
19 ---
20 kind: RoleBinding
```

```
21 apiVersion: rbac.authorization.k8s.io/v1
22 metadata:
23   name: foo-service-account-rolebinding
24   namespace: secure-ns
25 subjects:
26 - kind: ServiceAccount
27   name: foo-service-account
28   apiGroup: ""
29 roleRef:
30   kind: Role
31   name: foo-service-account-role
32   apiGroup: ""
33 ---
34 apiVersion: v1
35 kind: Secret
36 metadata:
37   name: foo-service-account-token
38   namespace: foo
39   annotations:
40     kubernetes.io/service-account.name: foo-service-account
41 type: kubernetes.io/service-account-token
```

Listing 5: ServiceAccount with permissions for fetching Pod resources

Listing 5 shows how to create a ServiceAccount token with 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 ServiceAccount, the Secret resource creates a similar authorization token as Pod with auto-mounting of ServiceAccount 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.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  spec:
4    replicas: 1
5    selector:
6      matchLabels:
7        app: app
8    template:
9      spec:
10       automountServiceAccountToken: false
11       containers:
12         - name: main
13           image: main-image
```

37

```
14        resources:
15          limits:
16            cpu: "2"
17            memory: 1024Mi
18        volumeMounts:
19          - name: service-account
20            mountPath: /var/run/secrets/kubernetes.io/
    serviceaccount
21      - name: sidecar-container
22        image: sidecar-image
23        resources:
24          limits:
25            cpu: "1"
26            memory: 512Mi
27    volumes:
28      - name: service-account
29        secret:
30          secretName: foo-service-account-token
```

Listing 6: Two container Pod with resource limits and manually mounted ServiceAccount

5.2.2 Enforcing the fields

Kubernetes' Admission controller is the go-to tool for preventing undesired configurations from entering the cluster. Unfortunately, the existing built-in hooks do not provide the option to enforce resource limits or manual ServiceAccount mounting. As there is no existing community solution, a custom-made ValidatingAdmissionWebhook needs to be created.

The GitHub development setup includes an example admission controller based on Slack's *simple-kubernetes-webhook* [64]. A template is supplemented with a custom validation function, *resource-validator*, which is shown in Listing 7. When a Pod specification with no `resource.limits` definition is validated, the if-statement on line 20 triggers, causing the validation to fail. The function can be trivially extended to allow CPU and memory values only in a given range.

```
1 package validation
2
3 import (
4   "github.com/sirupsen/logrus"
5   corev1 "k8s.io/api/core/v1"
6 )
7
8 type resourceValidator struct {
9   Logger logrus.FieldLogger
10 }
```

38

```go
11
12  var _ podValidator = (*resourceValidator)(nil)
13
14  func (n resourceValidator) Name() string {
15    return "resource_validator"
16  }
17
18  func EveryContainerHasResources(arr *[]corev1.Container) bool {
19    for _, container := range *arr {
20      if container.Resources.Limits == nil {
21        return false
22      }
23    }
24    return true
25  }
26
27  func (n resourceValidator) Validate(pod *corev1.Pod) (validation
      , error) {
28    if (!EveryContainerHasResources(&pod.Spec.Containers)) {
29      v := validation{
30        Valid:  false,
31        Reason: "all pod container do not have resources defined",
32      }
33      return v, nil
34    }
35
36    return validation{Valid: true, Reason: "resources defined"},
       nil
37  }
```

Listing 7: Go function for validating that resource limits are used

The validator for automatic mounting of ServiceAccounts could be implemented similarly. Instead of verifying resource limits, the function should verify that `PodSpec.AutomountServiceAccountToken` is set to false.

Custom Admission controller webhooks require a Deployment and a Service resource, as well as TLS certificates stored as a Secret resource. These resources are well-documented in the *simple-kubernetes-webhook* example. The example also provides a shell script for minting TLS certificates. However, in production environments, the TLS certificates should be generated by a specialized certificate manager workload installed in the cluster. Because of the effort needed to include production-ready webhooks in the demonstration setup, these are left as an exercise for the reader.

6 Network Isolation

Implementing a Zero trust network architecture in a Kubernetes cluster is not trivial, since building network isolation between containers in the same Pod is not possible with common CNI plugins and NetworkPolicies. 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 the ingress traffic can be easily identified with the destination port number, the source port for the egress traffic depends on the TCP implementation. As a result, NetworkPolicies cannot be used to manage egress traffic from the sidecar independently of the main container. For intra-Pod communication, traffic passes through the loopback device, which is created by a *loopback* plugin [12]. The plugin is not managed by the CNI plugin installed on the Nodes; instead, Kubernetes requires that the underlying container runtime provides the implementation. Most runtimes implement the reference CNI plugin [20], but, for example, CRI-O implements its own. Due to this, the plugin does not install with a controller that would enforce NetworkPolicies on the loopback device. Furthermore, since all intra-Pod traffic has source and destination IP addresses of `127.0.0.1`, the packets do not have clear identifiers that could be used for traffic filtering. This chapter introduces two general approaches to solving 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 isolation, the approach breaks the tight integration of sidecar containers to the application container. Most importantly, sidecars will have their own independent lifecycle and scheduling, and 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 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 8. The policies allow cluster and external traffic to the main application port 8888 while denying all other traffic to and from the Pod.

```
1 apiVersion: networking.k8s.io/v1
2 kind: NetworkPolicy
3 metadata:
```
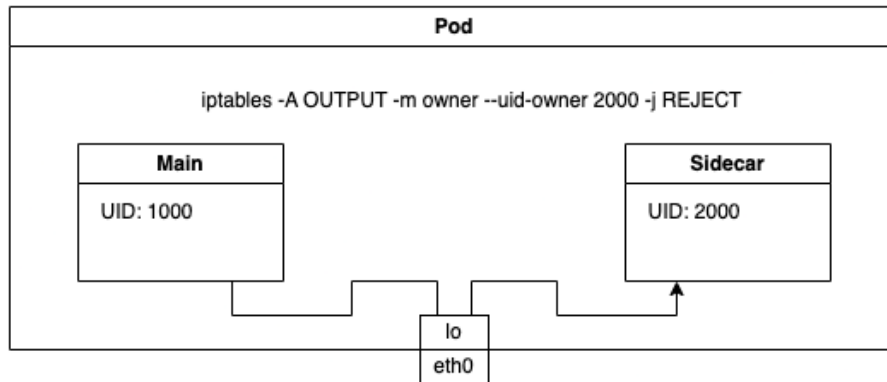
Figure 5: Single network namespace architecture

```
 4    name: deny-all
 5    namespace: app
 6  spec:
 7    podSelector: {}
 8    policyTypes:
 9      - Ingress
10      - Egress
11  ---
12  apiVersion: networking.k8s.io/v1
13  kind: NetworkPolicy
14  metadata:
15    name: web-allow-external
16    namespace: app
17  spec:
18    podSelector:
19      matchLabels:
20        app: node-app
21    ingress:
22    - ports:
23      - port: 8888
```

Listing 8: NetworkPolicy 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

the process's name and user, group, process, and session identifiers [40]. For example, `iptables -A OUTPUT -m owner --uid-owner 1000 -j REJECT` would reject all egress packets from containers with the specific 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 a unique user identifier can be filtered with IPTables.

Listing 9 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.

```bash
#!/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 9: 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 those originating from user id 1337, to the Envoy proxy [35]. 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 traffic control (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 10.

```
/* 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;
```

```
37    __u32 wire_len;
38    __u32 gso_segs;
39    __bpf_md_ptr(struct bpf_sock *, sk);
40    __u32 gso_size;
41    __u8  tstamp_type;
42    __u32 :24;     /* Padding, future use. */
43    __u64 hwtstamp;
44 };
```

Listing 10: skbuff struct in Linux v6.3 [38]

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. When combined with an IPTables rule that uniquely marks the packets egressing from the sidecar, for example, 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 to filter 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. Due to the above reasons, the solution relies only on IPTables rules for packet filtering.

6.1.2 Deployment

The commands for creating the IPTables rules could be applied to the Pod with an initContainer or postStart lifecycle handler. However, the commands require root user permissions with `NET_ADMIN` and `NET_RAW` capabilities and would require changing Pod's security admission level to privileged. Thus, it is more optimal to apply the rules outside the Pod context and Namespace; for example, by using a controller in an operator pattern. The same capabilities are paramount to any CNI plugin controllers that enforce NetworkPolicies using IPTables or eBPF.

For demonstration purposes, a custom container is used and installed with IPTables and `crictl` or `docker` CLI. The container uses the CLI tool to find the PID of a container in the Pod and uses `nsenter` to apply the IPTables rules inside the container's network namespace. The Dockerfile is found in Appendix C, and the DaemonSet is shown in the following Listing 11.

```
1 apiVersion: apps/v1
2 kind: DaemonSet
3 metadata:
4   name: iptables-injector
5   labels:
6     app: iptables-injector
7 spec:
```

```
 8    selector:
 9      matchLabels:
10        name: iptables-injector
11    template:
12      metadata:
13        labels:
14          name: iptables-injector
15      spec:
16        hostPID: true
17        containers:
18          - name: iptables-injector
19            image: iptables-injector
20            imagePullPolicy: Never
21            command: ["/bin/sh"]
22            # args: ["./iptables.sh"]
23            args: ["./iptables-file.sh"]
24            securityContext:
25              allowPrivilegeEscalation: true
26              readOnlyRootFilesystem: false
27              runAsUser: 0
28              runAsGroup: 0
29              runAsNonRoot: false
30              seccompProfile:
31                type: RuntimeDefault
32              capabilities:
33                add:
34                - NET_ADMIN
35                - NET_RAW
36                drop:
37                  - ALL
38              privileged: true
39            terminationMessagePath: /dev/termination-log
40            terminationMessagePolicy: File
41            volumeMounts:
42            - name: containerd-socket
43              mountPath: /var/run
44              readOnly: false
45        volumes:
46        - name: containerd-socket
47          hostPath:
48            path: /var/run
```

Listing 11: DaemonSet for a container injecting IPTables rules inside Pods

The script is not a valid controller as it does not implement a control loop that reacts to Pod creation. Instead, it only works when it is executed after the Pod has been

45

created. Therefore, it is only useful for demonstrating the concept and should not be used for purposes other than testing. Additionally, the DaemonSet requires extensive privileges: the CLI tool that is used to find the PID must be run on a privileged container that mounts the host's container runtime socket, and the `nsenter` command needs access to the host PID namespace.

6.2 Own network namespace for sidecar
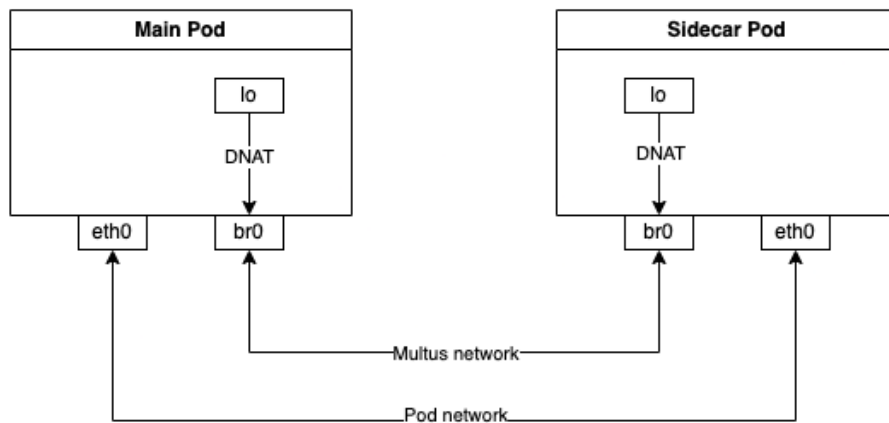
Figure 6: Multiple network namespaces architecture

The network architecture for the solution, where containers are deployed in separate Pods, is shown in Figure 6. The idea behind this approach is to utilize custom resources to regulate traffic between the containers, similar to NetworkPolicies. As containers are no longer considered sidecars, the solution must establish communication using 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 a separate IP address pool for the traffic, simplifying the re-implementation of loopback communication with a straightforward redirection rule.

6.2.1 Creating an 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 NetworkPolicies on top of the policy that implicitly denies all other traffic. The problem with this setup is that any traffic destined for `localhost` needs to be translated to Pod IP addresses and redirected to default Pod network. In comparison, when building new network interfaces for sidecar traffic with Multus, each Pod can have static IP addresses sidecar this traffic on a completely separate address pool.

```yaml
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 12: 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 13.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-app
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node-app
  template:
    metadata:
      labels:
        app: node-app
      annotations:
        k8s.v1.cni.cncf.io/networks: '[{
            "name": "localhost-replacement",
            "interface": "br0",
            "ips": [ "192.168.1.201/24" ]
          }]'
    spec:
      containers:
```

```
23        - name: node-app
24          image: node-app
25  ---
26  apiVersion: apps/v1
27  kind: Deployment
28  metadata:
29    name: node-app-statsd
30    namespace: app
31  spec:
32    replicas: 1
33    selector:
34      matchLabels:
35        app: node-app-statsd
36    template:
37      metadata:
38        labels:
39          app: node-app-statsd
40        annotations:
41          k8s.v1.cni.cncf.io/networks: '[{
42              "name": "localhost-replacement",
43              "interface": "br0",
44              "ips": [ "192.168.1.202/24" ]
45            }]'
46      spec:
47        containers:
48          - name: statsd
49            image: hypnza/statsd_dumpmessages
```

Listing 13: 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** [49], which mimics NetworkPolicies but only works for networks created by Multus. The resource is identical to NetworkPolicies, as seen in the Listing 14.

```
1  apiVersion: k8s.cni.cncf.io/v1beta1
2  kind: MultiNetworkPolicy
3  metadata:
4    name: deny-all-br0
5    namespace: app
6    annotations:
7      k8s.v1.cni.cncf.io/policy-for: localhost-replacement
8  spec:
9    podSelector: {}
10   policyTypes:
11     - Ingress
12     - Egress
```

48

```
13  ---
14  apiVersion: k8s.cni.cncf.io/v1beta1
15  kind: MultiNetworkPolicy
16  metadata:
17    name: allow-statsd
18    namespace: app
19    annotations:
20      k8s.v1.cni.cncf.io/policy-for: localhost-replacement
21  spec:
22    podSelector:
23      matchLabels:
24        app: node-app-statsd
25    ingress:
26      - from:
27          - podSelector:
28              matchLabels:
29                app: node-app
30        ports:
31          - protocol: TCP
32            port: 8125
```

Listing 14: 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* [50] and *multi-networkpolicy-tc* [52], both of which are created by the Multus team. As the names suggest, 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 traffic destined for the localhost to the new network. The script in Listing 15 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`.

```bash
1  #!/bin/bash
2  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 15: Script for mapping localhost to br0 interface

The demo solution on GitHub uses the same *iptables-injector* daemon container introduced in 6.1.2 for running the above script in container namespaces.

6.2.2 Co-scheduling the containers

The configuration described above does not replicate the behavior of traditional sidecars. The sidecar is specified in a separate Deployment resource, meaning it has a distinct lifecycle from the main container.

MutatingAdmissionWebhooks can be used to modify the Pod *CREATE* and *DELETE* actions so that the sidecar Pod is created and deleted in tandem with the main container. However, MutatingAdmissionWebhooks are designed only to operate on the resources sent to them. Kubernetes documentation states that any webhook that makes out-of-band changes or has any other side effects should be accompanied by a controller that periodically determines the actual state of the cluster [9].

A custom controller watching for a specific sidecar annotation could also implement co-scheduling; however, the controller must somehow keep track of which Pod the sidecars are assigned to. With this information, the controller could react to Pod lifecycle events, and fully support scaling, restarting, and cleanup of sidecar containers. Because of the complexity and the amount of work required to implement co-scheduling, this feature has been left out of the solution and is something to consider in the future.

7 Solution Evaluation

This chapter demonstrates with experiments that the solutions presented in Chapters 5 and 6 effectively isolate containers within the same Pod from one another. The evaluation not only verifies that the solutions meet the requirements but also examines their potential and any weaknesses that can be improved.

The experiments in the first section involve assessing security admission by deploying privileged Pods to the cluster, as well as enforcing network isolation by generating intra-Pod and inter-Pod HTTP traffic with client tools such as `curl`, `wget`, `kubectl` and `netcat`. The *malicious-sidecar* container (outlined in Appendix A) is added to the application containers as an additional sidecar and serves as an entry point for attacks in the experiments. The container has all the necessary tools installed.

```yaml
# Deployment configs...
spec:
  containers:
    # other application containers...
    - name: attacker
      image: malicious-sidecar
      command: ["/bin/sleep", "3650d"]
      imagePullPolicy: Never
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
        runAsUser: 2001
        runAsGroup: 2001
        runAsNonRoot: true
        seccompProfile:
          type: RuntimeDefault
        capabilities:
          drop:
            - ALL
      resources:
        limits:
          cpu: "1"
          memory: 256Mi
```

Listing 16: Configuration of the third, malicious container added to the deployments

Both network isolation solutions should only permit UDP traffic from the webserver container to the StatsD client. Any other successful request is deemed a failure in isolation. The second section examines the qualitative characteristics of the solutions, such as any observed limitations in security or usability when the solutions are used in production environments.

7.1 Security Evaluation

The experiments in this section verify that network isolation works correctly in both solutions. As a prerequisite for the experiments, the built-in Pod Security Admission rules should be enabled for the environments. To verify this, the first step in the evaluation is to deploy a Pod that is overly permissive into the cluster. The Deployment used to test this is shown in Listing 3. The definition namespace metadata fields were altered to match the namespace being tested, for example, app in our case.

When trying to deploy the Pod to the test cluster, a warning (illustrated in Listing 17) is displayed, which prevents deployment. This shows that Pod Security Admissions is functioning properly in the test environment, confirming the prerequisite, and allowing the evaluation to progress to testing network isolation.

```
1 Warning: would violate PodSecurity "restricted:v1.26":
2 privileged (container "sidecar-container" must not set
     securityContext.privileged=true),
3 allowPrivilegeEscalation != false (containers "main-application"
     , "sidecar-container" must set securityContext.
     allowPrivilegeEscalation=false),
4 unrestricted capabilities (containers "main-application", "
     sidecar-container" must set securityContext.capabilities.
     drop=["ALL"]),
5 restricted volume types (volume "host-root" uses restricted
     volume type "hostPath"),
6 runAsNonRoot != true (pod or containers "main-application", "
     sidecar-container" must set securityContext.runAsNonRoot=
     true),
7 seccompProfile (pod or containers "main-application", "sidecar-
     container" must set securityContext.seccompProfile.type to "
     RuntimeDefault" or "Localhost")
```
Listing 17: Warnings printed to terminal by Pod Security Admission

Experiments were conducted for both proposed solutions. The first solution, which uses a single network namespace, was tested with both Docker and containerd runtimes. However, since *multi-network-policy-iptables* does not work with Docker, the Multus solution was only evaluated with containerd runtime.

7.1.1 Experiment 1: sidecars cannot access main application

The first experiment tests the main application's network isolation from sidecar threats. As none of the sidecars in the system requires access to the main application container, isolation should block all requests from the sidecar containers. This can be tested by sending an HTTP request from the attacking or StatsD sidecar to localhost :8888 with *cURL*. If isolation is successful, the request will be rejected. This can be observed in the IPTables packet filter counters, which can be seen by using the iptables -xvn -L command.

The first solution successfully rejected packets when it was used with Docker runtime. However, when it was used with containerd, the packets were not rejected, even though the IPTables packet filter counters increased. This suggests that there is a discrepancy in how the CRIs implement the loopback CNI, and that containerd's implementation might be optimized to bypass certain elements of the network stack.

The second solution also rejected the packets as expected. Since it also implements the policies with IPTables, and containerd was used, this further implies that the unexpected result with the other solution is caused by how containerd implements the loopback CNI.

7.1.2 Experiment 2: only main application can access StatsD sidecar

The second experiment verifies that only the main application can send packets to the StatsD sidecar and that any other traffic directed there is blocked. To test packet filtering rules, UDP packets are sent from the *malicious-sidecar* to `localhost:8125` with netcat (`nc -u localhost 8125`).

The IPTables rules in the first solution reject the packets since the *malicious-sidecar* is deployed with a different user ID than the main application. Similarly, the IPTables rules generated from MultiNetworkPolicies in the second solution filter the unwanted packets.

7.1.3 Experiment 3: Access to other Pods is blocked

The third experiment verifies that neither of the network solutions interferes with NetworkPolicies applied outside the Pod network. The experiment environment uses a basic NetworkPolicy that blocks all outbound traffic and only allows inbound traffic to the main application. Policies can be tested by trying to access, for example, `google.com` or the Kubernetes API server from any container in the Pod. From outside of the Pod, only the main application port should be reachable.

The two solutions both work as anticipated and do not conflict with CNI's NetworkPolicies.

Table 4: Experiment results

Experiment	Solution 1, Docker	Solution 1, containerd	Solution 2, containerd
Isolated main	OK	Not OK!	OK
Isolated sidecar	OK	Not OK!	OK
Isolated Pod	OK	OK	OK

7.2 Qualitative assessment

The above experiments verify that the proposed solutions can be used to implement the desired network isolation. However, the solutions have several limitations and

should only serve as a proof of concept. These limitations should be mitigated before using the solutions in production environments.

The biggest limitation is that the solutions only function on specific container runtimes. Since the first solution has only been proven to function on the Docker runtime, it logically limits the potential places where it can be used. The limitation also raises the question of whether container runtimes should implement network isolation of sidecars. Since the runtimes are responsible for creating the loopback device and they appear to have varying implementations of it, it may be impossible to create a tool at the orchestrator level that is not dependent on the runtime.

Additionally, *multi-network-policy-iptables* does not function with Docker runtime. The issue for this is vaguely documented in the project's GitHub issues [51], but the issue was closed when Kubernetes dropped the explicit support for *dockershim* in version 1.24. Based on the experiments, the issue still exists on *cri-dockerd* setups. Furthermore, the documentation of *multi-network-policy-iptables* explicitly states that it has not yet been released as a stable version.

From a security point of view, the primary drawbacks are associated with the IPTables policies and the injector daemon. The first solution relies on distinguishing containers based on process user identifiers, which means that the system is only secure if the adversary cannot deploy containers with specific user IDs. The current implementation does allow the adversary to deploy a container with an arbitrary user identifier, which means that an adversary with permission to create containers can bypass isolation. Therefore, an admission controller is necessary to limit the allowed identifiers in production.

The injector daemon has the privileges to manage policies, so it should be regarded as a vital part of the cluster infrastructure, similar to CNI operators. The current container has root privileges and access to the Node's container runtime socket, which could be exploited to compromise the entire Node. To prevent this, the daemon's privileges should be reduced to the bare minimum. CNI plugins with operators could be used as a model for this enhancement. Furthermore, since the daemon does not function as a controller, the daemon needs to execute after application Pods are created to inject the IPTables rules. As such, solutions cannot handle normal container lifecycle events.

In conclusion, the solutions are intricate and need to be further enhanced before they can be employed in the production environment. The ideal solution would address the security issues mentioned above and implement a functioning controller loop to support container lifecycle events.

8 Discussion & conclusion

8.1 Future considerations

This chapter examines potential improvements to the proposed solution. As previously discussed, the proposed solution should not be used in production environments. A production-ready solution should incorporate a valid controller loop instead of the shell script that was proposed. Additionally, instead of relying on raw IPTables rules, the ideal solution should describe the policies with a custom resource, similar to NetworkPolicies. The controller loop should be responsible for managing both Pods and policy resources, similar to how many CNI plugins have been designed to accommodate NetworkPolicies. In contrast to common CNI plugins, the solution does not require the implementation of a CNI plugin, as the container runtime generates the loopback network interface. However, it is important to ensure that the selected container runtime does not require connectivity via the loopback device, which would be blocked by the applied policies.

8.2 Future of sidecars and service mesh

The release of Kubernetes version 1.28 in August 2023 brought with it the new "SidecarContainers" feature gate. This alpha-stage feature is designed to address the lifecycle and Pod termination issues that are currently experienced with sidecar containers [37]. Although existing Kubernetes primitives are capable of adequately managing sidecars, they are not sufficient for certain use cases [53]. For example, service mesh sidecars must be running and ready before other containers to ensure that all traffic is routed through the mesh. Similarly, if the sidecar terminates before other containers during shutdown, traffic from other applications may be lost. Additionally, lifecycle-related issues can impede log- and metric-collecting sidecars, forcing applications to resort to unconventional workarounds. Furthermore, in the event of an OOM error, the sidecar will not restart for jobs with `restartPolicy: Never`. This can render the application unreachable, as is the case with service meshes since the sidecar is necessary for application networking.

The introduction of sidecar containers as a new type of initContainer has made it easier to initiate and combine initContainers and sidecars in the start-up sequence. However, the proposal explicitly states as a non-goal that it does not allow the enforcement of different security regulations for sidecars compared to other containers. Therefore, it does not appear that the lowest security boundary will change from Pod to container in the near future. Nevertheless, the new distinction between sidecars and other containers could allow this development in the future.

From a security standpoint, the safest option is to avoid sidecars altogether. It is worth mentioning that the sidecar proxy is not a requirement for the implementation of a service mesh. For example, Cilium service mesh uses eBPF for network layer operations and an Envoy proxy per host for application layer operations such as L7 load balancing and policies [47]. For sidecarless mTLS, Cilium combines TLS handshake and network-level encryption with IPSec or Wireguard. This means that the service

mesh can actually operate without a proxy in some cases, and the proxy container is only used when necessary. The Envoy proxy is included in the cilium-operator, but since Cilium version 1.14, which was released in July 2023, the proxy can also be deployed as a DaemonSet independently.

Istio's service mesh also facilitates a sidecarless architecture. Ambient mesh [34], currently available as an open alpha version, uses ZTunnel proxies on each Node to provide L4 service mesh features such as mTLS, telemetry, authentication, and authorization. However, the ZTunnel is not capable of handling any L7 features, which are instead implemented by a waypoint proxy. The proxy operates on a per-namespace and per-service account basis, and it applies to all Pods in the same namespace by default. Sidecarless traffic encryption is enabled through encrypted HBONE tunnels.

Despite the advantages of sidecarless service mesh, it comes with its own drawbacks. Since all Pods on the Node use the same proxy, it can be subject to multi-tenancy issues such as the "noisy neighbor" anti-pattern, where one workload uses an excessive amount of resources and affects the performance of other workloads. Furthermore, the single proxy is a single point of failure, and if it fails, all Pods on the Node become unreachable. Additionally, the proxy is a more attractive target for attackers, as it can be used to impersonate any Pod on the Node.

The architecture of service meshes is a subject of much discussion in the industry, and the sidecarless design is not necessarily the next step in their development. In fact, the first service mesh, Linkerd, used the per-host proxy model in its version 1, but due to operational, management, and security issues, version 2 was based on sidecars.

8.3   Conclusion

This thesis provided an overview of containers, Kubernetes networking, sidecar design pattern, and Zero trust security architecture. It identified the security risks associated with sidecar containers in Kubernetes and showed how an adversary could exploit them. For issues that could not be resolved with existing tools, the thesis proposed two proof-of-concept solutions for creating network isolation between the application and sidecar containers. Although following best practices in hardening Kubernetes clusters can help, the architecture of Pods in Kubernetes makes it difficult to isolate sidecars. The first proposed solution implements network isolation within the Pod's network namespace using IPTables, while the second solution creates a network that mimics the shared network namespace of sidecars. Solutions demonstrate that it is possible to isolate sidecars; however, the complexity and special infrastructure required to overcome this architectural limitation quickly outweigh the benefits. From a security standpoint, tried and tested components are always preferable to custom infrastructure that requires maintenance.

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. *Dynamic Admission Control*. 2023. URL: https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers (visited on 08/15/2023).

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

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

[12] Kubernetes Authors. *Network Plugins*. 2023. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/ (visited on 07/29/2023).

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

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

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

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

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

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

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

[20] The CNI Authors. *Loopback CNI plugin source code*. 2023. URL: https://github.com/containernetworking/plugins/tree/v1.3.0/plugins/main/loopback (visited on 07/29/2023).

[21] 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.

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

[23] 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.

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

[25] 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.

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

[27] Michael Crosby. *What is containerd?* 2017. URL: https://www.docker.com/blog/what-is-containerd-runtime/ (visited on 08/15/2023).

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

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

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

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

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

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

[34] Istio developers. *Ambient Mesh Architecture*. 2023. URL: https://istio.io/latest/docs/ops/ambient/architecture/ (visited on 08/19/2023).

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

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

[37] Kubernetes developers. *Kubernetes v1.28 Changelog*. 2023. URL: https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.28.md#feature (visited on 08/24/2023).

[38] 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).

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

[40] Linux developers. *iptables(8) - Linux man page*. 2023. URL: https://linux.die.net/man/8/iptables (visited on 07/29/2023).

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

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

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

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

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

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

[47] Thomas Graf. *Cilium 1.14 – Effortless Mutual Authentication, Service Mesh, Networking Beyond Kubernetes, High-Scale Multi-Cluster, and Much More.* 2023. URL: https://isovalent.com/blog/post/cilium-release-114/ (visited on 08/19/2023).

[48] 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).

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

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

[51] Kubernetes Network Plumbing Working Group. *multi-networkpolicy-iptables: Issue in container-runtime=docker*. 2023. URL: https://github.com/k8snetworkplumbingwg/multi-networkpolicy-iptables/issues/1 (visited on 12/28/2023).

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

[53] Kubernetes Node Special Interest Group. *KEP-753: Sidecar containers*. 2023. URL: https://github.com/kubernetes/enhancements/tree/df246c8a3c89e1360ffa024f761e7ccc64d6e222/keps/sig-node/753-sidecar-containers (visited on 08/24/2023).

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

[55] 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).

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

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

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

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

[60] 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.

[61] hypnza. *hypnza/statsd_dumpmessages in Dockerhub*. 2023. URL: https://hub.docker.com/r/hypnza/statsd_dumpmessages (visited on 12/28/2023).

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

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

[64] Clément Labbe. *A Simple Kubernetes Admission Webhook*. 2022. URL: https://slack.engineering/simple-kubernetes-webhook/ (visited on 07/30/2023).

[65] 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.

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

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

[68] 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.

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

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

[71] 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.

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

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

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

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

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

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

[78] 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

```
1 FROM --platform=linux/amd64 ubuntu:22.04
2
3 RUN apt update
4 RUN apt install -y curl
5
6 # Setup kubectl
7 RUN curl -fsSLo /etc/apt/keyrings/kubernetes-archive-keyring.gpg
       https://packages.cloud.google.com/apt/doc/apt-key.gpg
8 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
9
10 # Install deps
11 RUN apt update
12 RUN apt install -y net-tools nmap ncat kubectl etcd iputils-ping
       iproute2
13
14 # kubeletctl
15 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
16
17 RUN apt install -y iptables
18
19 # Point to the internal API server hostname
20 RUN echo 'export APISERVER=https://kubernetes.default.svc' >>
     root/.bashrc
21 # Path to ServiceAccount token
22 RUN echo 'export SERVICEACCOUNT=/var/run/secrets/kubernetes.io/
     serviceaccount' >> root/.bashrc
23 # Read this Pod's namespace
24 RUN echo 'export NAMESPACE="$(cat ${SERVICEACCOUNT}/namespace)"'
       >> root/.bashrc
25 # Read the ServiceAccount bearer token
26 RUN echo 'export TOKEN="$(cat ${SERVICEACCOUNT}/token)"' >> root
     /.bashrc
27 # Reference the internal certificate authority (CA)
28 RUN echo 'export CACERT="${SERVICEACCOUNT}/ca.crt"' >> root/.
     bashrc
```

Listing 18: Dockerfile for penetration testing

B   Example webserver deployed with StatsD sidecar

```
1  const Koa = require('koa');
2  const app = new Koa();
3
4  const StatsD = require('node-statsd');
5  const client = new StatsD();
6
7  const logMessage = async (ctx, next) => {
8    await next();
9    const rt = ctx.response.get('X-Response-Time');
10   console.log('${ctx.method} ${ctx.url} - ${rt}');
11   client.timing('response_time', rt);
12   client.increment('response_counter');
13 }
14
15 const setResponseTimeCtx =async (ctx, next) => {
16   const start = Date.now();
17   await next();
18   const ms = Date.now() - start;
19   ctx.set('X-Response-Time', '${ms}ms');
20 }
21
22 app.use(logMessage);
23 app.use(setResponseTimeCtx);
24 app.use(async ctx => {
25   ctx.body = 'Hello World';
26 });
27
28 app.listen(8888);
```

Listing 19: Node.js server (`index.js`)

```json
{
  "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 20: package.json

```dockerfile
FROM node:18-alpine

WORKDIR /app

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

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

Listing 21: Dockerfile for the server

C Dockerfile for penetration testing

```
1 FROM --platform=linux/amd64 ubuntu:22.04
2
3 RUN apt update -y
4 RUN apt install -y iptables jq wget
5 # Install crictl
6 ENV CRICTL_VERSION="v1.26.0"
7 RUN wget https://github.com/kubernetes-sigs/cri-tools/releases/
    download/$CRICTL_VERSION/crictl-$CRICTL_VERSION-linux-amd64.
    tar.gz
8 RUN tar zxvf crictl-$CRICTL_VERSION-linux-amd64.tar.gz -C /usr/
    local/bin
9 RUN rm -f crictl-$CRICTL_VERSION-linux-amd64.tar.gz
10
11 COPY iptables.rules iptables.rules
12 RUN cat << EOF\
13 #!/bin/bash
14 APP_UID=1000\
15 APP_PORT=8888\
16 SIDECAR_UID=2000\
17 SIDECAR_PORT=8125\
18 set -x\
19 SIDECAR_PID=$(crictl -r unix:///var/run/containerd/containerd.
    sock ps --name statsd -q | xargs crictl -r unix:///var/run/
    containerd/containerd.sock inspect | jq '.info.pid')\
20 nsenter -t $SIDECAR_PID -n iptables-restore < iptables.rules\
21 set +x\
22 while true; do sleep 1; done\
23 EOF >> iptables-file.sh
24
25 RUN cat << EOF\
26 #!/bin/bash\
27 set -x\
28 MAIN_PID=$(crictl -r unix:///var/run/containerd/containerd.sock
    ps --name node-app -q | xargs crictl -r unix:///var/run/
    containerd/containerd.sock inspect | jq '.info.pid')\
29 nsenter -t $MAIN_PID -n sysctl -w net.ipv4.conf.all.
    route_localnet=1\
30 nsenter -t $MAIN_PID -n iptables -t nat -A OUTPUT -m conntrack
    --ctstate RELATED,ESTABLISHED -j ACCEPT\
31 nsenter -t $MAIN_PID -n 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\
```

```
32  nsenter -t $MAIN_PID -n iptables -t nat -A POSTROUTING -m
       addrtype --src-type LOCAL --dst-type UNICAST -j MASQUERADE\
33  set +x\
34  while true; do sleep 1; done\
35  EOF >> localhost-redirect.sh
```
Listing 22: Dockerfile for injecting IPTables rules

D Webserver and StatsD deployed with Multus networking

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

```yaml
42              readOnlyRootFilesystem: false
43              runAsUser: 0
44              runAsGroup: 0
45              runAsNonRoot: false
46              seccompProfile:
47                type: RuntimeDefault
48              capabilities:
49                drop:
50                  - ALL
51          resources:
52            limits:
53              cpu: "1"
54              memory: 1024Mi
55 ---
56 apiVersion: v1
57 kind: Service
58 metadata:
59   name: node-app
60   namespace: app
61 spec:
62   ports:
63   - port: 8888
64     protocol: TCP
65     nodePort: 30001
66   selector:
67     app: node-app
68   type: NodePort
```

Listing 23: 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:
              type: RuntimeDefault
```

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

Listing 24: 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:
```

```
46   name: allow-statsd
47   namespace: app
48   annotations:
49     k8s.v1.cni.cncf.io/policy-for: localhost-replacement
50 spec:
51   podSelector:
52     matchLabels:
53       app: node-app-statsd
54   ingress:
55     - from:
56       - podSelector:
57           matchLabels:
58             app: node-app
59       ports:
60         - protocol: TCP
61           port: 8125
```

Listing 25: Network interfaces and policies