Oksana Baranova

Multi-Tenant Isolation in a Service Mesh

Master’s Thesis
Espoo, December 31, 2021

Supervisor: Professor Tuomas Aura
Advisor: Sakari Poussa, M.Sc. (Tech.)
As the demand for cloud computing is increasing, more and more IT businesses migrate their services to the cloud. Containerization over virtualization is often used as it allows to run applications inside the container increasing security, portability, agility and faster delivery.

Open source communities develop new projects and add new functionalities to existing cloud technologies to improve cloud usage. Examples of such technologies are plenty. Kubernetes and Istio have become popular tools utilized to orchestrate containers and manage networks. However, with the growth of cloud technologies, the security issues become more and more severe and require additional security measures implemented.

A popular direction of deployment model in today’s K8s and Istio environment is multi-tenancy. It requires proper isolation mechanisms that will be considered in this thesis. Security requirements were designed based on a threat model which considers existing security threats in service meshes in multi-tenant environments.

The research proposes a certificate management solution for Istio and conducts experiments to test it. The designed solution contains a code for automated deployment and secure connection establishment. From the acquired results it can be concluded that the solution is able to address network-related threats and comply with performance requirements.

**Keywords:** cloud computing, Istio, Kubernetes, multi-tenancy, network security, service mesh

**Language:** English
Acknowledgements

I would like to thank professor Tuomas Aura for the review of the thesis and support throughout the thesis writing. In addition, I would like to thank the Aalto Admissions Committee for giving me the opportunity to study at one of the best universities in Finland. I learned a lot in the field of computer science, which gave me an understanding of how to develop professionally.

I wish to thank my manager Matti-Pekka Sivouso for bringing me to the service mesh team and supporting me during my internship at Intel. I wish to express my deep gratitude to my advisor Sakari Poussa for his valuable guidance, constructive suggestions and support for the thesis. I also want to say a big thanks to George Gaal for his thoughtfull explanations and answers to my countless questions.

I would like to thank my parents for their encourangment and emotional support while being thousands of kilometers away from me.

And in conclusion, I would like to thank myself for believing in me, dedication, and desire to learn.

Espoo, December 31, 2021

Oksana Baranova
Abbreviations and Acronyms

ABAC: Attribute-Based Access Control
AKS: Azure Kubernetes Cluster
API: Application Programming Interface
CA: Certification Authority
CNCF: Cloud Native Computing Foundation
CNI: Container Networking Interface
CPS: Certification Practices Statement
CPU: Central Processing Unit
CRI: Container Runtime Interface
CRD: Custom Resource Definition
CSR: Certificate Signing Request
DevOps: Development and Operations Methodology
DoS: Denial of Service
DNS: Domain Name System
HTTPS: Hypertext Transfer Protocol Secure
IT: Information Technology
K8s: Kubernetes
mTLS: mutual Transport Layer Security Protocol
NAT: Network Address Translation
OS: Operating System
OSI: Open Systems Interconnection Model
PKI: Public Key Infrastructure
RA: Registration Authority
RBAC: Role-Based Access Control
SaaS: Software as a Service
SGX: Software Guard Extensions
URI: Uniform Resource Identifier
VM: Virtual Machine
VPN: Virtual Private Network
ZTNA: Zero Trust Network Architecture
Contents

Abbreviations and Acronyms 4

1 Introduction 9
  1.1 Problem Statement ........................................ 10
  1.2 Objectives and methodology .............................. 11
  1.3 Thesis Outline ........................................... 12

2 Background 13
  2.1 Container Orchestration with Kubernetes ................ 13
     2.1.1 Architecture ........................................ 13
     2.1.2 Authentication and Authorization .................... 16
     2.1.3 Multi-Tenancy ...................................... 18
  2.2 Networks and Traffic Management ......................... 20
     2.2.1 Service ............................................ 20
     2.2.2 Ingress and Egress ................................... 21
     2.2.3 CNI ................................................ 23
     2.2.4 Network Policy ..................................... 23
     2.2.5 Istio Service Mesh ................................... 24
  2.3 Information Security ..................................... 27
     2.3.1 Network Security .................................... 27
     2.3.2 Isolation ........................................... 28
     2.3.3 Principle of Least Privilege .......................... 28
     2.3.4 Public Key Infrastructure ............................ 29

3 Solution Requirements 31
  3.1 Threat Modeling ......................................... 31
  3.2 Security Requirements ................................... 33
  3.3 Developer Environment ................................... 34
4 Multi-Tenancy Analysis

4.1 Scenario 1 / Services in different namespaces 36
4.2 Scenario 2 / Tenants in different namespaces 37
4.3 Scenario 3 / Network Policy 39
4.4 Scenario 4 / Istio Service Mesh with Authorization Policy 41
4.5 Scenario 5 / Istio Service Mesh with Multi-CA Support 43

5 Solution Evaluation 45

5.1 Security Evaluation 45
5.1.1 Experiment 1 45
5.1.2 Experiment 2 46
5.1.3 Experiment 3 47
5.1.4 Experiment 4 49
5.2 Performance Evaluation 51

6 Discussion 55

6.1 Single Mesh vs Multi-Mesh 55
6.2 Threats Mitigation 58
6.3 Recommendations 58
6.4 Summary 60

7 Conclusion 62

A Tenant’s Installation 70

B Multi-CA Installation 72
List of Tables

2.1 Kubernetes Components .............................. 15
3.1 Threat Modeling .................................. 32
3.2 Nodes Characteristics ............................... 35
6.1 Overview of Resources Isolation Options in Istio .... 56
6.2 Mitigation of Identified Threats ..................... 59
List of Figures

2.1 Kubernetes Architecture .................................................. 14
2.2 Control Plane Components Interaction During Cluster Resource Creation ........................................... 17
2.3 Single Tenant and Multi-Tenant Architectures ............... 19
2.4 Ingress ................................................................. 21
2.5 Routing based on URI path ........................................ 22
2.6 Hostname Routing ...................................................... 22
2.7 Network Policy Creation ............................................... 24
2.8 Istio Architecture ...................................................... 25

3.1 Multi-Tenancy Setup .................................................... 35

4.1 Scenario 1 ............................................................... 37
4.2 Scenario 2 ............................................................... 38
4.3 Scenario 3 ............................................................... 40
4.4 Scenario 4 ............................................................... 42
4.5 Multi-CA Architecture ................................................ 44

5.1 Access to the Same Namespace Service ......................... 47
5.2 Services in a Cluster ................................................... 48
5.3 Experiment 3 ........................................................... 49
5.4 Experiment 4 ........................................................... 50
5.5 Test Bed with Default Network Policy ......................... 51
5.6 Test Bed with Istio Proxy ............................................ 52
5.7 Measured Latency (50th percentile) ............................... 53
5.8 Measured Latency (75th percentile) ............................... 54
5.9 Measured Latency (99th percentile) ............................... 54

6.1 Multi-Mesh in a Single Cluster .................................... 57
6.2 Multi-Mesh in Multiple Clusters .................................. 57
6.3 Multi-Mesh with Multi-CA Support ............................... 58
6.4 Intel SGX ............................................................... 60
Chapter 1

Introduction

Current IT businesses adopt cloud computing technologies in order to accelerate the delivery of new features to end-users and increase profits by lowering overall costs. Cloud computing is a model for providing on-demand ubiquitous, convenient network access to a shared pool of abstract, virtualized computing resources such as networks, servers, storage, applications, and services [43]. These resources are dynamically scaled to adjust current workloads for optimal utilization and are typically provided on a pay-as-you-go basis.

With the advent of cloud technologies, microservice architectures have become very popular. It is a variant of a service-oriented software architecture aimed at interconnecting small, loosely coupled, and easily modifiable modules called microservices [21]. Each module performs a separate function and when necessary invokes other modules for other functionalities. When individual microservice crashes, new clones of this microservice can be spawned quickly due to their small size. This approach makes applications much more reliable, allowing efficient utilization of computing resources.

Service meshes are used for managing the microservices network when deploying applications [28]. A service mesh is a network structure that separates the control plane from the distributed data plane, which consists of additional proxies. Benefits of use include scalability of network management, observability of services communication, provision of secure connections, flexibility, and others [24].

Most cloud service providers offer multi-tenancy in order to benefit from the corresponding economies of scale, which also means savings for the end user. Multi-tenancy means that data and resources are located in the same computing instance, where users are given a certain level of control according to their needs. Multi-tenancy has made cloud computing popular, allowing businesses to benefit from lower costs while still accessing data and appli-
cations in the cloud [50]. However, there are a certain set of security risks associated with resource sharing and resource isolation in multi-tenant environments [30].

1.1 Problem Statement

With migrating microservices in the cloud, Kubernetes technology gained popularity as the best tool for container orchestration. At the same time, usage of the Istio service mesh to effectively manage service-to-service communication inside the K8s cluster has increased. K8s and Istio have become strategic components of the entire DevOps process, that is why attacks on them remain relevant. In the case of a compromised system, an attacker would gain access to all nodes and containers running, which is a direct path to compromise or leak confidential data.

K8s do not support multi-tenancy out of the box. The SIG Multi-tenancy of the K8s community is working on projects to ensure multi-tenant isolation on different layers of security [23]. However, there is no current project supporting multi-tenant certificate management in Istio Service Mesh. Users are not able to configure the workload certificate lifecycle within their service mesh. There are numerous reasons why users would wish to manage this - auditing, regulatory requirements and control.

One of the ways to achieve network isolation of tenants in a service mesh is to create a custom PKI system for a client in a shared cluster by blocking the mTLS communication between tenants. Thus, current work will consider this direction.

The motivation of the thesis lies in the lack of approaches for certificate management in a multi-tenant cloud environments whereas the demand for multi-tenant architectures is rapidly growing day by day. The suggested solution can be implemented in Istio service mesh environments deployed on top of K8s in order to build a secure multi-tenant network. A multi-CA support was added to Istio 1.12 release in December 2021 with the aim to enhance network security in K8s cluster. This thesis contributes to the development by providing usage scenarios of multi-tenant isolation in K8s with the focus on security and by evaluating the suggested solution based on performance.
1.2 Objectives and methodology

The primary goal of the thesis is to improve network security in a K8s multi-tenant environment using Istio service mesh without severely influencing the system’s performance.

The objectives of the thesis are as follows:

1. Conduct an overview of multi-tenancy, K8s, service meshes, and networks.
2. Identify security threats in a multi-tenant environment in service meshes.
3. Develop a solution to overcome the aforementioned challenges.
4. Conduct experiments to analyse multi-tenancy in K8s and Istio and to evaluate the proposed solution based on security and performance.

Research questions are as follows:

1. How to isolate different tenants during TLS communication potentially with the help of the certificate management process in a service mesh?
2. Is it reasonable and secure to use previous solutions in a multi-tenant environment such as authorization policies? Why is there a need for the proposed solution?
3. What limitations or disadvantages remain after the solution is implemented?

There are a number of scientific papers describing the security concerns in K8s, service meshes, and their application in the industry. However, these materials are not systematized and do not fully describe the problem of ensuring security in multi-tenant cloud environments. Based on the above information, it can be concluded that the topic under consideration has not been explored extensively.

Another aspect that needs to be considered is the research area is limited to network isolation, not taking into account other layers of security. However, it is essential to use best security practices for other layers in the hard multi-tenancy model as it implies a low level of trust between users.
1.3 Thesis Outline

This chapter (Introduction) presents motivation behind the research, relevance of the topic, methodology and problem statement. The rest of the thesis is structured as follows:

- Chapter 2 (Background) gives an overview of the concepts related to K8s, networking and service meshes.
- Chapter 3 (Solution Requirements) describes requirements for the solution based on a developed threat model, and provides information about the workstation and software used in this project.
- Chapter 4 (Multi-Tenancy Analysis) describes the experiments that were conducted showing how the network isolation works in K8s multi-tenant environment and service mesh.
- Chapter 5 (Solution Evaluation) evaluates the security and performance of the proposed solution.
- Chapter 6 (Discussion) discusses the proposed solution focusing on its advantages and disadvantages, and gives an overview of further improvements.
- Chapter 7 (Conclusion) concludes research findings.
Chapter 2

Background

2.1 Container Orchestration with Kubernetes

Kubernetes (K8s) is an open-source technology developed by Google and used to automatically orchestrate containerized applications including their deployment, scaling and management. It allows to create a cluster of many servers and provides an automated control over application execution and resource use, thereby, relieving the support team from routine tasks and avoiding human errors.

Currently, K8s is the most popular technology for orchestrating containerized applications [45], and it is supported by many open-source communities across the world. On 18 October 2021, it had more than 3100 contributors, and over 100 000 commits [13]. Special Interest Groups (SIGs) were created to support developers and operators to improve the technology. SIGs focus on different aspects, such as multi tenancy, bare metal, networking, storage, and other operational areas.

2.1.1 Architecture

K8s cluster architecture consist of master node and worker nodes. Master node stands for the cluster control plane which has control components to manage other worker nodes, whereas worker nodes are the planes of container’s execution and launch, or in other words, servers on which containers are running (Figure 2.1). To provide high cluster availability and fault tolerance, multiple master nodes can be used.

When the kubectl command-line tool is executed, it is directed to the kube-api server on a master node, which is a gateway for coming requests. Master node has four main components running: kube-api-server, kube-controller-manager, kube-scheduler and etcd. Each of them as its own functionality.
Worker node, in its turn, has two main processes: *kubelet* and *kube-proxy*. The description of the components is presented in Table 2.1.

To orchestrate containers, K8s provides different types of objects that are registered as cluster API endpoints, such as Pods, Deployments, ReplicaSets and others. A Pod is a unit of deployment and container abstraction of a K8s cluster that consist of one or more logically combined containers. Basically, Pod is a set of Linux features such as namespaces, cgroups, and others [17]. K8s assign IP addresses to Pods and a single DNS name for a set of Pods that change during the re-creation of a Pod entity. A Service is a unit of abstraction that combines a set of Pods into a single entry point. In general, communication between Pods can be divided into several types of communication [9]:

1. Between strongly connected containers in the same Pod (communicate with each other via the loopback interface).

2. Between Pods (Pod-to-Pod communication).

3. Between the Pod and the Service (Pod-to-Service communication).
CHAPTER 2. BACKGROUND

<table>
<thead>
<tr>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kube-api-server</td>
<td>cluster gateway that handles coordination of all processes in a cluster including authentication, authorization and processing stages</td>
</tr>
<tr>
<td>kube-controller-manager</td>
<td>main controller that performs functional management by ensuring that the current state of the system matches the desired state</td>
</tr>
<tr>
<td>kube-scheduler</td>
<td>component that schedules workloads for each deployed component and performs resource allocation on these servers</td>
</tr>
<tr>
<td>etcd</td>
<td>key-value store that contains information about cluster state and from what other components take information</td>
</tr>
<tr>
<td>kubelet</td>
<td>component that listens for workloads scheduling on the kube-api server, deploy workloads on nodes and updates the kube-api server with information about general status, Pods availability, resource usage</td>
</tr>
<tr>
<td>kube-proxy</td>
<td>component responsible for request forwarding and providing communication between Pods and Nodes</td>
</tr>
<tr>
<td>container runtime</td>
<td>software that is responsible for running containers</td>
</tr>
</tbody>
</table>

Table 2.1: Kubernetes Components

4. Communication with Services outside the K8s cluster.

To be able to run K8s Pods, a container runtime should be installed into each Node of a cluster. The container runtime is responsible for container execution and supervision and is connected to K8s via CRI [34]. When the kubectl command is executed against K8s cluster to create a Pod, the request comes to the kube-api-server that forwards it to kube-scheduler. The scheduler checks the information about Nodes availability and Pods specification for certain conditions and decide what Pod should be located on what Node. Then, kubelet receives the information from the scheduler and creates a container. The resource status is being updated in the etcd by adding information about the container that is running in a particular Pod. K8s associates information with the Pod resource after the launch of the container.

Container runtime stack consists of three layers: client, high-level runtime, and low-level runtime. Kubelet communicates with a high-level container runtime and creates a container runtime interface (CRI). The CRI is a plugin interface which enables kubelet to use a wide variety of container run-
times [14]. High-level runtime manages images accessed from a registry and hands them over to a low-level runtime that launches the containers using Linux features. The most popular container runtime environment is Docker that provides application isolation and immutability with the use of images and their declarative description. When Docker service is started, it creates a virtual network interface `docker0` with a virtual IP address range where all containers are grouped. Thus, it allows external access and communication between containers.

### 2.1.2 Authentication and Authorization

K8s technology uses a declarative management approach, where controllers implement a control loop [10], by constantly comparing the desired state of a cluster with the current one and by making updates when necessary. The `kube-api-server` provides CRUD (create, read, update, delete) interface for querying and changing the state of a cluster via RESTful APIs, storing this state in the `etcd` key-value storage. Moreover, this component is responsible for authorization and authentication of clients.

In order to communicate with the K8s cluster, users send their requests to the control plane via an Application Programming Interface (API), Command Line Interface (CLI) tool, or a Web User-Interface (Web UI) Dashboard. Each access request goes through the authentication, authorization, and admission control stages. The detailed interaction between control plane components when creating a new resource is shown in the sequence diagram in Figure 2.2.

Any process inside or outside of the cluster should be authenticated when making requests to the `kube-api-server` [34]. The authentication stage in K8s exists for two categories of users: service accounts and normal users. While service accounts are managed by K8s API, normal users cannot be added to a cluster through an API call. Several methods are used for the authentication such as static token files, bootstrap tokens, service account tokens, OpenID connect tokens, passwords, X.509 client certificates [7]. In this thesis’s experiments, X.509 client certificates and service account tokens are used. Any user that presents a valid certificate signed by the cluster’s CA is considered authenticated.

After the authentication has proceeded, the authorization stage takes place that opens access to the granular division of access rights to certain resources and divide the network into segments. Authorization occurs on one or more grounds, which are usually described in resource access policies. The different authorization modes are used in K8s to permit or deny the API requests [8].
In the ABAC mode, the ABAC authorizers are used to grant access to API requests, which combine policies with attributes. To enable this mode, the API server can be started with the command: 

```
{authorization-mode=ABAC option.
```

An authorization in the RBAC mode is based on users roles. When creating a K8s Role object, resource access is restricted by specific operation, such as create, get, update, patch, and so on. The Role grants access to resources within a specific namespace, whereas the Cluster Role grants access with a cluster-wide scope.

Once the Role is created, the K8s Role Binding object is used to bind users to the same namespaces as a Role. Cluster Role Binding object is used
to grant access to resources at a cluster-level to all namespaces.

After the API requests are authenticated and authorized, admission controllers are used to specify access control policies, which include allowing privileged containers, checking on resource quota, and others. An admission controller is a piece of code that intercepts requests to the K8s API server prior to persistence of the object, but after the request is authenticated and authorized [18]. These execute the mutating and validating admission control webhooks which are configured in the API. In the case if the request is rejected by one of the controllers, the error returns immediately to the end-user.

2.1.3 Multi-Tenancy

Multi-tenancy is a mechanism that allows multiple users (or tenants) to access one software instance (or shared component) [38]. In the context of Kubernetes, multi-tenancy is defined as hosting workloads owned by more than one user on the same shared K8s cluster. Tenants can be defined as a group of users, or teams in an organization, e.g. the development team. Multi-tenancy effectively addresses shared resources and contention concerns by leveraging specific K8s features and recommended guidelines for maintaining workloads [50]. In addition, it ensures a proper isolation of running workloads and implements security policies and features to control and limit access to only those resources that a given user should have access to.

In a single tenant per cluster architecture, each tenant is dedicated to its own K8s cluster. In this case, everyone has their own resources whereas other tenants are completely unaware of other clusters and its components. In a multi-tenant per cluster architecture, cluster resources are shared among multiple tenants (Figure 2.3). The definition of a tenant used in this thesis is given in Section 3.2 of Chapter 3.

Multi-tenancy has two forms: soft and hard. The Soft Multi Tenancy model means that there is a low probability of a potential security breach happening while tenants trust each other. Thus, the cluster has no strict isolation rules. It can be applied to an organization using a multi-tenant K8s environment, where tenants have their own resources and no access to others. In the case of an access policy violation, an employee can be held responsible. On the other hand, the Hard Multi Tenancy model enforces strict isolation rules in order to prevent potential security breaches. Tenants cannot trust each other and consider themselves as potential attackers. In this case, an organization can involve people from other organizations, implying the need for resources isolation [49].
The concern for multi-tenancy appears when workloads negatively impact other workloads by:

- consuming too many resources and causing resource starvation;
- accessing and using components that they should not have access to;
- using cluster without safeguards in place to promote safe sharing.

The demand for the multi-tenancy is increasing. There are plenty of reasons. Firstly, in single tenant per cluster architecture, it can be challenging to efficiently operate multiple K8s clusters, especially if clusters are located in different geographical regions. Secondly, some resources such as policy controllers, monitoring, logging should be consistent. Thirdly, obviously there is a need to share resources between tenants. The benefits of having a multi-tenant system outweigh its drawbacks. Sharing a cluster is much more efficient for resource management, however, additional security measures are required to achieve isolation.
CHAPTER 2. BACKGROUND

2.2 Networks and Traffic Management

Secure routing firewalls, VPNs, and other network virtualization technologies are used to securely manage client traffic with network-level encryption ensuring the confidentiality of the data being transferred over a transmission channel. There are several ways to manage traffic between Pods and external connections in a multi-tenant K8s environment.

2.2.1 Service

Service is a mechanism to combine several Pods into a single abstract entity and to provide uniform access to them [6]. By default, Pods can communicate with each other without NAT. Non-Pod processes on a Node, such as kubelet, are able to communicate with all Pods presented on that Node [9]. There are different types of Services used in K8s:

1. ClusterIP is the default Kubernetes service. It provides a Service within the cluster that can be accessed by other applications within the cluster. There is no external access.

2. Headless Service is a subtype of the ClusterIP Service that does not imply load balancing. Instead, the type of Service name is returned in records containing the IP addresses of the Pods [1].

3. NodePort is a type of Service that allocates a specified port on each Node of the cluster, traffic from which will be proxied to the Pods. The Service is accessed using a request to the address NodeIP:NodePort.

4. LoadBalancer is a type of Service that makes a Service available outside the cluster using an external load balancer provided by the cloud provider where the cluster is located.

5. ExternalName - the type of Service that allows creating a record CNAME [9] which will be returned when requesting a Service. In this case, the request will be processed at the DNS level, and not using proxying or redirection.

The main advantages of using services include ability to execute at the kernel level (when using ClusterIP or NodePort), routing of internal and external traffic and built-in traffic load balancing. The disadvantages are in the lack of built-in implementation to work with TLS, lack of routing of requests at the application level, lack of ability to trace requests and in limiting only external traffic for the cluster.
CHAPTER 2. BACKGROUND

2.2.2 Ingress and Egress

Egress filtering is a term used to describe the filtering of network traffic that leaves a host. This type of filtering is commonly used to control communications from a private network to public networks, but it is rarely used on a private network. This happens for several reasons. Outbound filtering requires more consideration of how hosts intend to communicate. Ingress filtering is generally considered good enough to stop unwanted communication on the network. Outbound filtering requires knowledge of each expected flow, which is not usually found in traditional networks.

Ingress is a type of resource in a K8s cluster that provides HTTP and HTTPS routes from outside the cluster to services in the cluster. The Ingress is managed by Ingress controller. Ingress can be configured to provide external URL services, load balancing, SSL/TLS traffic termination, and hostname-based virtual hosting. The main purpose of Ingress is to provide more powerful HTTP-based routing to route traffic based on URL parts (Figure 2.4).

![Figure 2.4: Ingress](image)

Traffic routing is controlled by rules defined by Ingress. There are several types of rules:

1. Routing to host by default. All traffic coming to this Ingress is redirected to the service specified in the defaultBackend rule unless another host is specified.

2. Routing along the URI path. Traffic that matches the rule will be redirected to the specified service. A rule can be based on either a complete match or a prefix (Figure 2.5).

3. Hostname routing. Routing is done through the hostname in the request. Multiple domain names can refer to the same IP address assigned
to the Ingress, thus representing shared hosting. The host rule can be specified as a wildcard (Figure 2.6).

Figure 2.5: Routing based on URI path

Figure 2.6: Hostname Routing

The benefits of using an Ingress include filtering external traffic by hostname in an HTTP request, filtering external traffic along the path in an HTTP request and built-in implementation of work with TLS. However, Ingress can only process requests based on HTTP/HTTPS protocols and network blocking applies only for traffic external to the cluster.
2.2.3 CNI

CNI stands for Container Networking Interface that is used to create a generic plugin-based networking solution for containers [39]. Basically, it is a simple interface between container runtime and networking implementation. Initially, containers lack a network interface, and Kubelet calls a CNI plugin to perform network operations on a container and provides network configuration and container data to the plugin. The CNI plugin is used in K8s to integrate a network interface into the container network namespace and assign an IP address to the interface. It sets loopback and eth0 interfaces for a particular container. The CNI plug-in controls Pods on subnets on Nodes, and updates the routing rules in the network namespace of each Node [44].

Currently, the CNI is a CNCF project that consist of three main components:

- CNI specification: the API between runtimes and network plugin.
- Conventions: extensions to the API that are not required for all plugins.
- Libraries: a Go implementation of the CNI specification that plugins and runtimes can use.

There are plenty of container runtimes used by CNI, such as Docker, Podman, Apache Mesos, Amazon ECS and others. Network plugins used by CNI are Calico, Flannel, Cilium, Azure CNI and many others [11].

2.2.4 Network Policy

K8s has a built-in traffic management tool called network policies used for bidirectional traffic filtering to restrict both inbound and outbound connections in a cluster. This can impact items from network filtering through to how Pods interact with the underlying host. The entities that the Pod can interact with are identified by a combination of three identifiers:

1. Other pods, communication with which is allowed (Pod cannot restrict access to itself). The rule is determined based on a Pod selector.

2. Allowed namespaces. The rule is determined based on a namespace selector.

3. Blocks of IP addresses (except for the IP address of the node where it is running). The rule is determined by a subnet or a list of IP addresses.

The sequence diagram of the network policy creation process is shown in Figure 2.7.
2.2.5 Istio Service Mesh

For more convenient and configurable management of communications between services in the cluster, service meshes are used. The main functions include service discovery, smart load balancing, metrics collection, and request authorization [37]. Service mesh allows to reduce development costs due to the transfer of the request processing logic from the application to the proxy server (Sidecar Proxy) and the possibility of reuse regardless of the application programming language.

The Istio open-source service network is one of the most widespread, since it was originally created for implementation in K8s clusters, and is actively developed and supported by the community [5]. The main components of the Istio service network are illustrated in Figure 2.8. The service
network architecture is subdivided into two layers: Data Plane and Control Plane. The control plane provides a single interface for configuring the service network and tracking requests policies and configuration for all proxies operating at the data plane within the service network. The control plane contains the Istiod component that consists of Mixer, Pilot, and Citadel components previously being separately managed (in earlier releases of Istio) and is responsible for [37]:

- validating and applying configurations;
- discovery of services in the cluster and intelligent traffic management according to the existing configuration;
- controlling access to resources and collecting telemetry data on traffic movement within the cluster.

In order to intercept traffic from services and to enforce admin-defined policies Istio places its own proxy based on Envoy proxy (Sidecar Proxy). The Envoy is an intermediary application between the service and external
users which redirects traffic to Istiod to check access or send metrics, or balances the load between replicas.

The proxy server performs the following tasks [35]:

- Discovery of the services available for communication for a given application and checking its health.

- Routing of requests at the transport (L4) and application (L7) layers of the OSI model.

- Load balancing when there are multiple instances of related services.

- Implementation of the circuit breaker pattern to prevent an application from failures.

- Implementation of the operation repeat pattern to repeat a request in the case of unsuccessful response from an external service.

- Two-way encryption of the connection using mTLS allows not only to protect the transmitted data from interception but also to control the ability to establish a connection between applications.

- Ensuring observability by logging and tracing traffic.

The benefits of using Istio include limiting external and internal K8s network traffic, built-in implementation of TLS (mTLS), traffic control at the application level, automatic logging and traffic tracing, load balancing, implementation of circuit breaker and operation repeat patterns, and many others.

The disadvantages of using a service mesh are as follows:

1. Increasing consumed computing resources to run a proxy server for each instance of application;

2. Reducing speed of processing requests by adding additional links in the request chain.

3. Incompatibility of the interaction protocol with traffic routing at the transport layer and the lack of implementation of traffic routing at the application layer;

4. Lack of restriction of access to the target application (solved by restrictions at the application level).
CHAPTER 2. BACKGROUND

2.3 Information Security

The advent of microservice architecture opens up opportunities for developing, building, testing, and deploying system components separately. As the system grows and the provided functionality increases, the number of components and environments for development rises. To ensure the proper level of security, appropriate tools for traffic management should be used to avoid data leakage, followed by loss of reputation and money for businesses. As the thesis focuses on network isolation, this section will consider network security.

2.3.1 Network Security

The challenging problem of network security is to ensure secure communication between entities of the communication process. K8s is a highly networked system with many communications protocols used. However, many of these protocols operate traffic with sensitive information, such as cluster secrets or credentials, and lack the strongest security mechanisms. Therefore, enforcing HTTPS, providing Ingress and Egress filtering for clusters, and using the strongest cryptographic controls should be enforced to secure a network in a cluster.

The aforementioned mechanisms (described in Section 2.2) are complex enough to manage in production clusters one by one, thus, service meshes are used. Istio uses the mutual TLS (mTLS) protocol by default to authenticate components against each other. mTLS is the variation of the standard TLS protocol used to authenticate both sides of the communication process instead of one [27]. The protocol is advantageous in terms of certificate life cycle management and transparent encryption. The information about certificate and its usage is presented in the following subsection of this chapter.

A Zero Trust Network Architecture (ZTNA) is an approach to the design of IT systems where network components are not trusted by default. The approach supports mutual authentication, including identity verification and integrity of devices without regard to location [32]. Istio Service Mesh is used to create a zero-trust network by providing granular access to network resources and ensuring a secure connection even in the case of a physical compromise of the data transmission channel [22].

The basis of the zero-trust model is the assumption that attackers located everywhere, and protection is needed not only from outside world, but also between system components. This corresponds to the hard multi-tenancy model in cloud environments, where tenants are not trusted, as a result of
which they need to be isolated from each other. Multi-tenancy isolation in this thesis implies consideration of the network isolation only.

2.3.2 Isolation

Isolating deployed applications by limiting the set the resources they can access is important in a zero-trust network. Applications have traditionally run in a shared environment, where user applications run in a runtime environment with very few restrictions on how those applications can interact. This shared environment poses a high risk in the event of a compromised application and creates problems similar to the perimeter model [31]. Application isolation aims to limit the damage done by a potentially compromised application by clearly defining the resources available to the application. Isolation will limit the capabilities and resources that the operating system provides:

- processor time;
- memory access;
- network access;
- access to the file system;
- access to system calls.

Application isolation can be accomplished through various technologies, such as SELinux, AppArmor, BSD jails, Virtualization, Containerization, Windows Isolated Applications, and many others. In the thesis, only applications that use isolation through containerization will be considered.

2.3.3 Principle of Least Privilege

Any object of the information system should only be granted the necessary rights required to perform its certain functions. In the case of applications, this usually means running it under a dedicated user, in a container, or using some other means of isolating the application. It is important to understand which actions require which privileges so that they can only be granted when needed [36].

In a zero-trust network, users must similarly surf the network in limited privilege mode most of the time, raising their permissions only when necessary to perform some sensitive operations. For example, an authenticated
user can freely access a version control system. However, access to a criti-
cal production system requires additional confirmation that the user or his
system has not been compromised.

For relatively low-risk activities, this privilege escalation can be as simple
as requesting the user’s password, a second factor token, or sending a push
notification to the user’s phone. For high-risk access, an active acknowledg-
ment from the peer via an out-of-band request can be requested.

Applications should be configured to have the minimum amount of priv-
ileges required to operate the network. Unfortunately, applications deployed
in a production environment are often given wide access over the network
that opens up additional threats for the system. Thus, ensuring the prin-
ципle of minimum privileges in modern information systems is a relevant and
important aspect of information security.

2.3.4 Public Key Infrastructure

A PKI is a set of several components that works together to create, manage,
distribute, use, store and revoke digital certificates. The goal of the PKI
is to enable secure communication between parties over the Internet. Many
protocols, such as HTTPS, SSL are based on a successful PKI implementation
that provides non-repudiation, privacy, integrity, and trust [25].

Public key cryptography involves asymmetric cryptography with a secret
private key and a published public key. These keys are paired with the each
other in the certificate but stored separately. PKI ensures the confidentiality
of the messages being transferred as they are encrypted using a public key.
The decryption of the messages is possible with a corresponding private key.
In addition, the integrity of information is achieved by creating a digital
signature of the message with the use of the private key. A receiver can
verify the authenticity of the sender based on the certificate being used [2].

A certification authority (CA) is the trusted authority that certifies indi-
viduals identities and create digital certificates based on the information
about individuals to ensure confidentiality, authentication, and data integrity
[26]. The digital certificate establishes an association between the subjects
identity and a public key. Every CA should have a certification practices
statement (CPS) that defines how identities should be verified including
generation, maintenance and transmission of certificates. In addition, it de-
scribes how keys are secured, what data is places in a certificate, and how
revocation procedure should be handled. Apart from CA, a registration au-
thority (RA) is often used to verify the identity of parties requesting their
digital certificates and forward requests to a CA. However, it is not used for
issuing and signing the certificates.
CHAPTER 2. BACKGROUND

The crucial part of the PKI is the trust between end-users and CAs. However, some accredited CAs can be compromised and issue fake certificate that bind a domain name to a key pair held by attackers, not by legitimate communication endpoints. Google proposed the certificate transparency (CT) to detect fraudulent certificates [40]. It has been widely adopted by CAs, websites, and TLS libraries [33].

A certificate signing request (CSR) is the actual request to a CA containing a public key and information for the certificate generation. A digital certificate binds an individuals identity to a public key including all information needed for receiver to assure the authenticity of public key owner. After the RA verifies an identity, the CA generates the digital certificate based on the X.509 standard that defines the format of public key certificates. Usually, an X.509 digital certificate includes [25]:

- Version number - identifies the version of the X509 standard [51] that was used to create the certificate.
- Serial Number - provides a unique number that identifies the certificate.
- Subject - specifies the owner of the certificate.
- Public Key - identifies the public key being bound to the certified subject and algorithm that was used to create the key pair.
- Issuer - identifies the CA that generated and signed the certificate.
- Validity - specifies the period of time the certificate is valid for use.
- Certificate usage - specifies the approved use of the certificate and intended use of the public key.
- Signature Algorithm - identifies the hashing and digital signature algorithms used to sign the certificate.
- Extensions - allows additional data to be encoded into the certificate to expand the functionality of the certificate.

In K8s, users can be authenticated against kube-api-server with TLS certificates. The key and certificate itself are generated using the configured CA. After the certificate is verified, the CommonName and Organization fields from the Subject field of the certificate are stored in a temporary structure for authorization.
Chapter 3

Solution Requirements

This chapter analyzes network-related vulnerabilities existing in K8s cluster and defines security requirements for the solution in order to mitigate identified threats.

3.1 Threat Modeling

A threat is defined as a potential loss of confidentiality and integrity of an asset caused by an attacker acting to compromise that asset via some attack surface. Threat modeling is the process to optimize the information security of the system by identifying targets and vulnerabilities that can be exploited without countermeasures taken to prevent or mitigate the impact of threats to the system [47]. The process is important for a risk analysis and further product security activities plan.

Usually, there are three main categories of threats possible in K8s environment. Firstly, the access control and isolation category imply only authorized tenants are able to get access to certain resources. Secondly, the probability of side-channel attacks exists as non-authorized tenants can try to retrieve the information about the system based on memory and CPU resource consumption, for example. They can potentially bypass access control rules and exploit virtualization vulnerabilities, such as DoS attacks that are possible in K8s with the use of privileged containers. The last category is misconfiguration of K8s itself leading to information leakage. All of those form a base attack surface for the attacker for exploitation.

The threat model was developed in the scope of multi-tenant isolation in a service mesh based on K8s Security Assessment conducted by SIG Security in 2019 [29]. Identified threats, their description and actions to mitigate are shown in Table 3.1.
## CHAPTER 3. SOLUTION REQUIREMENTS

<table>
<thead>
<tr>
<th>Threat</th>
<th>Description</th>
<th>Actions to Mitigate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading K8s secrets and listing resources belonging to other tenants</td>
<td>An attacker can intercept internal traffic using sniffing programs and get the sensitive information.</td>
<td>Use of transparent TLS encryption for cluster resources.</td>
</tr>
<tr>
<td>Istio CA private key is stored unencrypted in K8s secret</td>
<td>A malicious tenant with administrator rights can read the key.</td>
<td>Use of Intel SGX technology to hide the Istio CA private key.</td>
</tr>
<tr>
<td>Theft of Issued Certificates</td>
<td>An attacker with access to a certificate and key can authenticate to the kube-api-server as any tenant.</td>
<td>Use of encrypted storage for CAs.</td>
</tr>
<tr>
<td>Side channel attacks and/or limits and requests are not assigned</td>
<td>A malicious tenant can gather information about api-resources belonging to another tenant based on information gathered from K8s system. No assigned limits and requests could lead to container escape issues.</td>
<td>Use of API priority and fairness and SGX security model (trusted execution environment).</td>
</tr>
<tr>
<td>Human mistakes in configuration (network/authorization policies without PKI)</td>
<td>Developers or cluster admins can misconfigure policies that will lead to unauthorized access to cluster components from third parties.</td>
<td>Use of multi-CA instead of policies to avoid probability of misconfiguration.</td>
</tr>
<tr>
<td>Unauthorized access to the kube-api server</td>
<td>No controls limiting Egress and Ingress to the kube-apiserver itself.</td>
<td>Use of Envoy sidecar with Egress policies in Istio with mTLS by default.</td>
</tr>
</tbody>
</table>

Table 3.1: Threat Modeling
3.2 Security Requirements

In next thesis context, tenant is defined as a set of users that have shared access to a given Istio control plane and full control of any workloads accessible by the tenant. A tenant’s associated users may not have admin control over the control plane, beyond setting specific configurations. Tenants should not have access to other tenants workloads and configurations. A tenant cannot see discovery information and internal names from a different tenant. Public names exposed over DNS to the Internet are visible the same for other tenants and internet users [46]. Configurations made by one tenant should not impact any other tenant especially in the case of high load or misconfiguration.

Main categories of possible actors in Istio cluster are as follows:

- Control Plane Cluster Administrator. A user that has complete control over a cluster running the control planes. In a multi-tenant SaaS environment [48], this would be the service provider for Istio as a service.

- Mesh Administrator. A user that performs tasks related to installation and maintenance of an Istio control plane on behalf of a tenant. A Mesh Administrator has access to a well-defined set of config options that are supported, as well as debug tools that respect the Isolation requirements.

- Mesh Security Administrator. A user that configures mesh-wide security settings (e.g. which TLS ciphers to use) for a mesh, and can also authorize inter-mesh connections.

- Mesh User. A user capable of running workloads in a namespace in a dedicated or shared cluster.

Workloads isolation means a workload behaves in the same way as if it was running in a completely separate cluster. A tenant cannot see configurations or endpoints from other tenants. A workload cannot directly talk with workloads from another tenant (which would imply a tenant can discover internal discovery info from another tenant).

Namespace isolation means for a single tenant, a set of exportTo and Sidecar.egress configs that define a compatible and consistent subset of tenant isolation. Two tenants cannot communicate with each other (except using explicit Gateways, with same controls used for external internet).

The solutions should properly maintain mTLS protocol with a reliable and flexible certificate management system. Thus, tenants might need to use independent root CAs.
The main questions for the solution provided would be:

1. How to limit communication between services using one CA or different CAs?

2. How to ensure the certificate provisioning path is valid?

3. How to ensure high availability of the solution? Is it rational to use multi mesh?

### 3.3 Developer Environment

Multi-tenancy analysis was performed in a K8s cluster running on the OpenStack platform (version 18/Wallaby) with one master node and one worker node with Intel Ice Lake processors. The platform is a free cloud computing platform deployed as infrastructure-as-a-service (IaaS) in both public and private clouds where virtual servers and other resources are made available to users. The software platform controls pools of compute, storage, and networking resources throughout a datacenter, all managed and provisioned through command-line tools, or through RESTful web services. It ensures high availability of applications deployed on it by providing additional components for orchestration, fault and service management [19].

Each node represents a separate VM joint in a cluster with the characteristics shown in Table 3.2.

Calico CNI (version v3.21.0) was chosen as a network plugin that provides L3 network solution. It sets up a bridge between Pod’s network namespace and network namespace of the host.

Istio Service Mesh (version 1.11.3) was chosen for the solution design as it is the most suitable service mesh having security mechanisms and integration with other technologies used for the certificate management process.

Cert-manager addon was used for certificate management process (version v1.5.4) as it automates the certificate management and issuance of TLS certificates from various issuing sources, ensures the validity of certificates and renews them if necessary before they expire [4]. The tool is well-integrated with Istio and is able to sign CSRs from workloads.

The development setup is shown in Figure 3.1
### Table 3.2: Nodes Characteristics

<table>
<thead>
<tr>
<th>unit</th>
<th>master node</th>
<th>worker node</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>istio-3-control-plane-4m2gf</td>
<td>istio-3-md-0-wxrg6</td>
</tr>
<tr>
<td>role</td>
<td>control-plane,master</td>
<td>worker</td>
</tr>
<tr>
<td>k8s version</td>
<td>v1.20.5</td>
<td>v1.20.5</td>
</tr>
<tr>
<td>internal-IP</td>
<td>192.168.3.70</td>
<td>192.168.3.171</td>
</tr>
<tr>
<td>external-IP</td>
<td>10.237.71.219</td>
<td>10.237.71.215</td>
</tr>
<tr>
<td>OS-image</td>
<td>Ubuntu 20.04.2 LTS</td>
<td>Ubuntu 20.04.2 LTS</td>
</tr>
<tr>
<td>kernel</td>
<td>5.4.0-70-generic</td>
<td>5.4.0-70-generic</td>
</tr>
<tr>
<td>container-runtime</td>
<td>containerd:1.4.4</td>
<td>containerd:1.4.4</td>
</tr>
</tbody>
</table>

---

**Figure 3.1: Multi-Tenancy Setup**

K8s cluster

worker node

ns: bar

Alice

istio-proxy

kubelet

control plane

kube-api-server

ns: foo

Bob

istio-proxy

developer machine

source code

git push

github.com

git pull

kubectl
Chapter 4

Multi-Tenancy Analysis

As was discussed in earlier chapters, K8s is not an ideal technology for multi-tenancy, thus, additional security measures are required. This chapter contains a step-by-step demonstration of multi-tenant network isolation in K8s cluster achieved with network security mechanisms (described in Chapter 2). Initially, the cluster contains only K8s system components such as coredns, etcd, kube-apiserver, kube-proxy, kube-scheduler, Calico CNI and openstack-controller. One scenario follows another one based on security limitations found.

Tenants were assigned to different K8s namespaces in order to isolate their resources. For example, a Service inside the bar namespace belongs to tenant Alice whereas a Service in the foo namespace refers to tenant Bob. Tenant’s installation details is provided in Appendix A. In further scenarios, Istio service mesh is added as a default mechanism for network isolation.

All manifests used for the multi-tenancy analysis and the evaluation of the suggest solution with related materials can be found at GitHub public repository https://github.com/oksanabaranova/Master-s-thesis.

4.1 Scenario 1 / Services in different namespaces

Current scenario examines a connectivity between Services in different namespaces without isolation mechanisms implemented. Pods as basic units of K8s are ephemeral and after a restart, they change their IP addresses. When using a Service, a user sends a request to one IP address of the particular Service, and the Service redirects it to a certain Pod. As Services in current scenario have a cluster IP service type, they are reachable only within a cluster. In order to reach a Service from external entity, Node Port or Load
Balancer types should be used.

Simple applications httpbin and sleep were deployed in the cluster in different namespaces foo and bar. From the results, Services were able to reach each other as was expected (Figure 4.1). The reason is a K8s’s default realization for Service-to-Service communication if not restricted by other network security policies and as long as the Pods/Services IP addresses are known [9]. When creating a Service, cluster admin or user should specify labels for a Service to select Pods under that label. When Service is applied, all Pods under the given label refers to that particular Service.

![Figure 4.1: Scenario 1](image)

4.2 Scenario 2 / Tenants in different namespaces

In this scenario, two tenants Alice and Bob were created and assigned to namespaces bar and foo respectively with httpbin and sleep workloads. To authenticate them against a cluster, X509 Client certificates were used (client certificate authentication) with an openssl tool. After the certificate approval by K8s administrator, those users were added to the cluster with assigned RBAC rules (Figure 4.2).

When a user context is assigned to the existing tenant in a kubeconfig file, the kubectl command utility run commands based on permissions given to this particular tenant [34]. According to the RBAC rules assigned, some tests were conducted to ensure isolation between tenants is provided:
Figure 4.2: Scenario 2

```bash
$ kubectl get role -n bar
NAME    CREATED AT
alice   2021-10-04T12:22:26Z
$ kubectl get role -n foo
NAME    CREATED AT
bob     2021-10-04T12:22:26Z
$ kubectl create deployment -n bar --image nginx name1 --as=alice
error: failed to create deployment: deployments.apps is forbidden: User "alice" cannot create resource "deployments" in API group "apps" in the namespace "bar"
$ kubectl get services -n bar --as=alice
NAME TYPE CLUSTER-IP     EXTERNAL-IP PORT(S) AGE
httpbin ClusterIP 10.101.57.173 <none> 8000/TCP 4h49m
sleep ClusterIP 10.97.224.39 <none> 80/TCP 4h49m
$ kubectl get services -n bar --as=bob
Error from server (Forbidden): services is forbidden: User "bob" cannot list resource "services" in API group "" in the namespace "bar"
$ kubectl get deployments -n foo --as=bob
NAME READY UP-TO-DATE AVAILABLE AGE
httpbin 1/1 1 1 4h51m
sleep 1/1 1 1 4h51m
$ kubectl delete deployment sleep -n foo --as=bob
Error from server (Forbidden): deployments.apps "sleep" is forbidden: User "bob" cannot delete resource "deployments" in API group "apps" in the namespace "foo"
```
RBAC rules given to tenants Alice and Bob ensure the isolation of defined resources between tenants, however, Services in different namespaces still reach each other (Figure 4.2). To isolate them, the K8s Network Policy is usually used and it will be considered in the next section.

### 4.3 Scenario 3 / Network Policy

The basic principle of the zero-trust model is to provide the principle of least privilege. In the case of network communication, in order to ensure minimum privileges, only the traffic that is necessary for the operation of the application is allowed. All other traffic should be denied. The allowed traffic can be divided into two types:

- Infrastructure - provides access to the infrastructure components of the cluster (DNS server, monitoring application).
- Application traffic - traffic that provides business functions for an application deployed in a K8s cluster.

In current scenario, the Network Policy was applied to the cluster in order to isolate applications within a namespace. In the initial configuration of a zero-trust network, traffic of the bar namespace accepts only traffic inside the bar namespace. All Ingress and Egress traffic outside of the namespaces was prohibited by applied Network Policies:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: bar-ingress-egress-deny
  namespace: bar
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: foo-ingress-egress-deny
  namespace: foo
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```
CHAPTER 4. MULTI-TENANCY ANALYSIS

The results have shown that applied Network Policy successfully blocked communication between Services httpbin and sleep as they are belong to different tenants Alice and Bob (Figure 4.3).

![Diagram of K8s cluster with httpbin and sleep services in different namespaces](image)

**Figure 4.3: Scenario 3**

Network policies are used for simple network traffic filtering. However, filtering rules are based on labels that cause an additional threat of direct connections to the application ports. A malicious tenant can bypass Ingress resources and Load Balancer setup when accessing an open port. Moreover, K8s do not warn users when policies are applied, which may not be enforced without further configuration changes or components. For example, Network Policy requires a CNI that can process and accept policies as configuration. The user will not be notified that the policy has not been applied, leading to a false sense of security.

Another issue is related to the scalability of network policies. When network policies scale to hundreds of namespaces and thousands of modules, these resources quickly become unmanageable due to the complexity of the hierarchy being built, the rules in which can overlap and overwrite each other, leading to security holes. In addition, network policies lack some functionality such as using a Gateway for the internal traffic. Thus, the Istio service mesh is considered as a primary solution for the aforementioned problems.
4.4 Scenario 4 / Istio Service Mesh with Authorization Policy

By default, Istio CA generates a self-signed root certificate and key and uses them to sign workload certificates. In addition, Istio supports custom CA integration with the K8s CSR. In this case, Istiod acts as the RA that authenticates and authorizes the CSRs. However, only one signer is available for signing certificates, which is a limitation when it comes to multi-tenancy and services isolation [41].

A default K8s network policy can also be used to isolate services in a service mesh, however, it has the drawback of lacking a granularity for specific endpoints. Therefore, additional filters have to be used to block undesired communication between services. A Sidecar object of Istio restricts services that proxy can reach when forwarding traffic from workloads. An Authorization Policy provides access control for certain workloads in a service mesh.

In addition, there is a problem with X509 certificates. In the case of a private key stolen, making certificate revocation procedure could be hard to perform. A cluster admin should make a certificate revocation procedure by revoking a CA and regenerating all issued by this CA certificates. Thus, in some cases the use of service accounts is recommended to avoid an issue because service account token can be easily replaced in the case of emergency situation.

In this scenario, services httpbin and sleep in the namespace bar are isolated from services httpbin and sleep in the namespace foo by additional filters, such as ExportTo, Sidecar, Authorization Policy. Authorization policies were applied to the namespaces bar and foo to restrict access to each other:

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-sample
  namespace: bar
spec:
  action: DENY
  rules:
  - from:
    - source:
      namespaces: ["foo"]
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
```
name: deny-sample
namespace: foo
spec:
  action: DENY
  rules:
  - from:
    source:
      namespaces: ["bar"]

From the experiment results, it can be clearly seen that network isolation is ensured with additional filters (Figure 4.4). However, these filters become problematic to manage at a large scale when cluster has many workloads and namespaces. Authorization policies can overwrite each other and be easily misconfigured when the number of security rules increases.

Figure 4.4: Scenario 4
4.5 Scenario 5 / Istio Service Mesh with Multi-CA Support

In this scenario, multiple CAs are added to Istio with the help of cert-manager tool to approve and sign K8s CSR from workloads which are deployed in Istio. Services deployed in different namespaces can be isolated without additional filters (such as discussed in Section 4.4) but with different CAs instead. It means that workloads hold different CA. These CAs are able to sign workload certificates with a particular namespace to block network connectivity between services via mTLS.

The cert-manager was installed in the cluster. Then, CA Issuers were created by generating keys and certs and applying a manifest file. In order to integrate Istio with a cert-manager, the Istio Operator was installed with a reference to ClusterIssuer names. The detailed installation and commands used for it are shown in Appendix B.

Figure 4.5 illustrates the multi-CA architecture. There are two namespaces bar and foo containing httpbin and sleep services. Each service that is willing to authenticate itself against Istio, would request a CSR approval from a particular issuer before signing a certificate. Certificate issuer is specified in CSR in such a way that services in different namespaces are assigned to different certificate issuers. The signing controller validates the signing conditions, creates a certificate and updates the CSR. Istio agent receives namespace information from a Pod running under the service, and certificate signer domain information in order to differentiate between isolated entities. Signers foo and bar can deny CSRs in the case of certain conditions are not met.

After all, tenants with the same signers are able to communicate between each other, whereas tenants with different issuers are not. The solution implies confidentiality and integrity of the data transferred over a transmission channel and authenticity of sides participating in the communication process.
Figure 4.5: Multi-CA Architecture
Chapter 5

Solution Evaluation

This chapter evaluates security and performance of the suggested multi-CA solution in Istio service mesh (that was described in Section 4.5). The motivation behind the evaluation is to show the solution in action from the security and performance sides. Tests in Security Evaluation section investigate how well the multi-CA solution blocks undesired communication between services in different namespaces foo and bar that belong tenants Alice and Bob. Experiments in Performance Evaluation section help to understand whether the multi-CA solution worth to implement without degrading the performance in comparison to Authorization Policies that serve the same function in Istio.

5.1 Security Evaluation

5.1.1 Experiment 1

In this experiment, Services isolated by multi-CA solution from different tenants were tested for connectivity between each other. As a result of the experiment, the solution successfully blocked the communication between Services separated by different CAs responsible for issuing workload’s certificates for different namespaces. Each kubectl requests coming to particular Service lacks the certificate issued by the same namespaced CA. For example, the web-page content of the httpbin service did not appear, thus, the Service was not reached. The reason is the TLS connection error that was caused by certificate verification process:

```bash
$ export SLEEP_POD_FOO=$(kubectl get pod -n foo -l app=sleep -o jsonpath={.items..metadata.name})
kubectl exec -it $SLEEP_POD_FOO -n foo -- curl http://httpbin.bar:8000/html
```
The proposed solution has an in-built feature to compare service’s CAs with the `istioctl proxy-config` command:

```bash
$export HTTPBIN_POD_FOO=$(kubectl get pod -n foo -l app=httpbin -o jsonpath={.items..metadata.name})
$export HTTPBIN_POD_BAR=$(kubectl get pod -n bar -l app=httpbin -o jsonpath={.items..metadata.name})
$istioctl proxy-config rootca=compare $HTTPBIN_POD_FOO $HTTPBIN_POD_BAR
```

Error: Both [httpbin-55cbddbc77-rbvx.foo] and [httpbin-55cbddbc77-jz74x.bar] have the non identical ROOTCA, theoretically the connectivity between them is unavailable.

### 5.1.2 Experiment 2

In second scenario, the `curl` command was executed to reach `httpbin` service located in `foo` namespace inside a default directory of the `sleep` service container belonging to the same namespace. A simple script was executed for services to connect to each other. According to experiment results, services in same namespaces were able to reach each other with the 200 status code (HTTP OK) whereas services in different namespaces were not (503 status code (Service Unavailable)):

```bash
$for from in "foo" "bar"; do for to in "foo" "bar"; do
  kubectl exec "$(kubectl get pod -l app=sleep -n $from -o jsonpath={.items..metadata.name})" -c sleep -n $from --
curl -s "http://httpbin.$(to):8000/ip" -s -o /dev/null -w "sleep.$(from) to httpbin.$(to): %{http_code}\n" done;
done
```

The `httpbin` service has been reached as the content of the `httpbin` webpage appeared (Figure 5.1):

```bash
$kubectl exec -it $SLEEP_POD_FOO -n foo -- curl http://httpbin.foo:8000/html
```
5.1.3 Experiment 3

By default Istio prohibits all incoming traffic to the service mesh until an Istio Ingress Gateway object with routing rules is installed. The Gateway is a Custom Resource Definition (CRD) in K8s, defined after installing Istio in K8s cluster and activating the ability to specify the ports, protocol and hosts for which the incoming traffic is allowed. It is binded to the Virtual Service in its specification field with a name ”gateways”. Istiod distributes configuration changes all over the proxies that are specified as a result of custom routing rules.

In this experiment, httpbin service in foo namespace was exposed to the external world through the use of Gateway and Virtual Service objects. The Virtual Service defines custom traffic routing rules applied to the service mesh. In this case, it routes request coming from any host to the httpbin.foo with the html subpath through the Gateway. The configuration of the installed Gateway and Virtual Service objects is as follows:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: external-service
  namespace: foo
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
```

Figure 5.1: Access to the Same Namespace Service
In order to check the connectivity, the external IP address of the Istio Ingress Gateway was retrieved, and the request to the `httpbin.foo` service was sent from another VM. In current setup, the range of IP addresses in `10.237.71.0/24` refer to the public Internet. The list of services presented in the cluster with IP addresses is shown in Figure 5.2. From the experiment results, the request to the `httpbin.foo` resource was denied because the certificate verification failed. This is due to the fact that `httpbin.foo` service is isolated by a namespaced CA, and accepts only traffic coming from workloads with the same CA (Figure 5.3).

![Figure 5.2: Services in a Cluster](image)

The experiment showed that external communication with tenant’s services can be easily blocked by the multi-CA solution.
5.1.4 Experiment 4

Istio supports secure communication for Gateways objects [6]. The objects can be configured for single or multiple hosts or be extended to support mutual TLS. In this experiment, a secured Istio Ingress Gateway was added to the existing setup to allow traffic to the isolated by CAs services. The configuration of the Gateway object was changed to support mTLS connections with a reference to K8s secret containing private key and certificate issued by Bar and Foo CAs of the gateway. It is needed for the Ingress Gateway to retrieve unique credentials corresponding to a specific "credentialName" field:

```yaml
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: gateway-tls
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: https-foo
      protocol: HTTPS
tls:
  - mode: MUTUAL
    credentialName: gateway-secret-foo
    hosts:
      - foo.example.com
      - port:
        number: 443
```

![Figure 5.3: Experiment 3](image-url)
The experiment showed that requests coming to Services through the Istio Ingress Gateway with certificates issued by appropriate CAs are accepted. The mTLS connection allows to authenticate both entities of communication process increasing the network security in the service mesh. The \textit{foo.example.com} is accessible to clients with the certificate issued by \textit{Foo CA} (Figure 5.4).

Access to the \textit{foo} Service is given when presenting the \textit{foo} client credentials containing the information about client key, client certificate, and certificate of the CA in the \textit{curl} request:

```bash
curl --noproxy "*" --cacert foo/tls.crt --key ./foo/client.key --cert ./foo/client.crt -v --resolve "foo.example.com:443:${INGRESS_IP}" https://foo.example.com(headers)
```
5.2 Performance Evaluation

The test bench was deployed in Azure cloud provider environment to minimize costs and overhead of setting up and configuring the cluster. An AKS cluster consist of 3 nodes, two of them include control plane components and workloads, whereas a third one is used for benchmarking purposes. The reason of the test bench isolation is to avoid interference of memory and CPU resources between cluster components and test tool. The suggested multi-CA solution is compared with authorization policies and network policies. The diagram of the test bench is shown in the Figure 5.5 and Figure 5.6.

The test bench contains a client `wrk2` and service `httpbin` used as a server. Both client and server are deployed in the same namespace `foo` to minimize latency. The description of the test bench is as follows:

- Cloud provider - Azure Kubernetes Cluster [3].
- Service `httpbin` acting as a server.
- Client application for benchmarking - `wrk2` [20].

![Figure 5.5: Test Bed with Default Network Policy](image)

Load testing was used as a technique for measuring and comparing response time latency. Traffic from the client application to the server was limited by service mesh using a proxy server depending on the cluster configuration. The measurements were carried out with the following parameters:

- One container used one virtual core (1 thread).
- Number of requests per second in a range 500-5000.
- 120 seconds of the test duration.
CHAPTER 5. SOLUTION EVALUATION

- Traffic control method: Istio with authorization policy (mTLS), Istio with multi-CA support (mTLS) and default K8s network policy without mTLS.

- Delays for percentiles 50, 75 and 99 were used as criteria.

Based on the results of load testing, a comparative analysis was carried out to assess the dependence of the latency on the traffic control method. The latency was measured taking into account delayed starts. The results are shown in Figure 5.7, Figure 5.8 and Figure 5.9.

Based on a comparative analysis of the dependence of the latency on the traffic control method, the thesis infers the following:

1. Use of default network policies without mTLS minimizes the impact on response time delays.

2. Use of Istio with mTLS shows significant difference in latency when processing more than 1000 requests per second.

3. A minimum registered delay in traffic proxying is 31 ms.

4. Authorization policy applied in the default Istio and multi CA solutions do not differ from each other according to the test results. Both solutions have relatively the same latency.
Taking into account the results of a comparative analysis of traffic control methods in K8s described in Chapter 2 and a comparative analysis of the response time delay, the thesis concludes the advantage of using a multi CA solution over authorization policies, since this approach allows to minimize the time spent on traffic limiting and do not require the allocation of additional computing resources. An K8s administrator would create a trusted CA to isolate services within a namespace belonging to different tenants instead of writing multiple authorization policies.
Figure 5.8: Measured Latency (75th percentile)

Figure 5.9: Measured Latency (99th percentile)
Chapter 6

Discussion

6.1 Single Mesh vs Multi-Mesh

As was discussed in previous chapters, the single tenant per cluster model is often used when the need for tenant’s isolation increases. However, in the edge computing scenarios, resources are very limited and cannot be easily spread among different clusters. Therefore, service meshes can be used in multiple models, each one has its own advantages and disadvantages:

- Single Mesh, Single Cluster - one Istio mesh is used in a single cluster. Usual approach that is the most popular.
- Single Mesh, Single Cluster via multi-CA - one Istio mesh is used in a cluster with multiple CAs for isolated workloads. The new solution that has been accepted by Istio community.
- Multi-Mesh, Single Cluster (Figure 6.1) - several Istio meshes are used in a cluster.
- Multi-Mesh, Multi-Cluster (Figure 6.2) - several Istio meshes are used in several clusters.
- Multi-Mesh, Multi-Cluster with multi-CA support (Figure 6.3) - several Istio meshes are used in several clusters, services are isolated from each other by namespaced CAs.

A comparison between proposed available models of how service meshes can be used is presented in Table 6.1.

A Multi-Mesh, Multi-Cluster option implies the combination of multiple service meshes across K8s clusters. This is a decentralized approach in which
<table>
<thead>
<tr>
<th>Option</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Mesh, Single Cluster (no Multi-CA)</td>
<td>native upstream support</td>
<td>to isolate resources additional filters are needed; more config changes; lack of identity isolation</td>
</tr>
<tr>
<td>Single Mesh, Single Cluster with multi-CA</td>
<td>different root CAs for tenants; no need for additional filters; no additional config changes when new workload is added</td>
<td>additional custom CA is needed</td>
</tr>
<tr>
<td>Multi-Mesh, Single Cluster</td>
<td>different root CAs for tenants; no need for additional filters; no additional config changes when new workload is added</td>
<td>patched solution; upgrade concerns; lack of identity isolation</td>
</tr>
<tr>
<td>Multi-Mesh, Multi-Cluster</td>
<td>native multi-tenant support</td>
<td>maintenance effort; more resource consumption; lack of identity isolation</td>
</tr>
<tr>
<td>Multi-Mesh, Multi-Cluster with multi-CA</td>
<td>enhanced multi-tenant support with better network security</td>
<td>additional custom CA is needed</td>
</tr>
</tbody>
</table>

Table 6.1: Overview of Resources Isolation Options in Istio

Each cluster uses its own control and data layer. Two or more clusters can participate in the service mesh regardless of the region or cloud provider. The advantage of the mesh aggregation model is the ability to selectively link between clusters and provide services from one cluster to another.

A Multi-Cluster approach is required for a high availability and fault tolerance of a K8s cluster. For example, compromised Istiod can be quickly removed from the cluster setup without influence on other cluster components because another Istiod is running separately from the affected one. The Federation model is introduced as the configuration of multiple clusters with a master host managed through a single API [15]. Unfortunately, the approach requires more administration overhead and additional expenditures in comparison to Single Mesh.

Thus, Single Mesh in a Single Cluster with multi-CA is advised for more efficient resource management, strict network isolation, and relatively low cost of maintenance.
CHAPTER 6. DISCUSSION

Figure 6.1: Multi-Mesh in a Single Cluster

Figure 6.2: Multi-Mesh in Multiple Clusters
6.2 Threats Mitigation

As was mentioned in Section 3.1, the threat modeling process identified several threats in the scope of multi-tenancy in Istio service mesh. The threat mitigation process implies actions to prevent or remediate identified threats. After the conducted experiments, the thesis infers that the multi-CA solution for tenant’s isolation addresses some of the identified threats (Table 6.2).

6.3 Recommendations

Multi-Tenancy is about fair resource management and isolation between tenants. In order to mitigate all of the identified threats, the isolation should be considered at all layers: OS, network, application, storage, and others. However, as it was mentioned in previous chapters, proposed solution considers network isolation only.

As K8s technology rely on containerization, security of containers become an important aspect of protection. An attacker can take advantage of misconfigured containers and compromise them in a cluster, which indirectly increases the compromise or includes application vulnerabilities. Container compromises include manipulation of internal switching, process control, or file system access. Untrusted application inside a container can lead to DoS attacks and resource exhaustion.

To implement OS layer isolation, K8s administrator should start with container images. Images should be tested for performance and quality because untrusted images can execute commands and access additional resources. Artifacts needed to build images should be version controlled. To avoid re-
<table>
<thead>
<tr>
<th>Threat</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading secrets and listing resources belonging to other tenants</td>
<td>Mitigated as multi-CA uses mTLS for tenant’s authentication by default and all unauthorized requests are denied.</td>
</tr>
<tr>
<td>Reading Istio CA key</td>
<td>Not mitigated as proposed solution relies on network security only without consideration of low-level security. Additional security measures are required, such as Intel SGX.</td>
</tr>
<tr>
<td>Theft of Issued Certificates</td>
<td>Not mitigated as the risk of CA compromisation exists, and certificate revocation procedure is not fully implemented in K8s. Encryption of the CA’s storage or use of service accounts is recommended.</td>
</tr>
<tr>
<td>Side-channel attacks and/or limits and requests are not assigned</td>
<td>Not mitigated as the multi-CA solution works on a network layer only. Use of best K8s security practices, API priority and fairness, and SGX security model are advised to mitigate the possibility of side-channel attacks occurrence.</td>
</tr>
<tr>
<td>Human mistakes in configuration (network/authorization policies without PKI)</td>
<td>Mitigated as the need for policies configuration no longer required.</td>
</tr>
<tr>
<td>Unauthorized access to the kube-api server</td>
<td>Mitigated as Istio uses TLS by default.</td>
</tr>
</tbody>
</table>

Table 6.2: Mitigation of Identified Threats

source exhaustion, it is essential to identify workload resource requests and limits [42].

Pod Security Standards are used to achieve low-level security in K8s. There are different policy types identified that range from highly restricted to highly flexible [16]:

- Privileged: unrestricted policy which gives unlimited rights to Pods. The most dangerous type.
- Baseline: a policy that restricts access at a minimal level by preventing privilege escalation problems.
- Restricted: a restricted policy that follows best security practices for Pods hardening.
To provide better protection of K8s secrets containing private keys for CAs, the use of Intel’s SGX technology can be used. The SGX is a system of architectural enhancements defined to help protect application integrity and confidentiality of data, and to withstand certain software and hardware attacks. It provides a trusted computing enclave where data (e.g., private keys) and applications (e.g., CA) are protected independently of the OS or hardware configuration (Figure 6.4). The trusted computing platform can be implemented in a multi-tenant environment to prove the authenticity of the platform.

![Figure 6.4: Intel SGX](image)

6.4 Summary

Answers for the questions (described in Chapter 3, Section 3.2) regarding the solution would be:

1. The communication between Services in Istio using one CA can be implemented with the use of different filters, such as Sidecars, ExportTo, Authorization Policies whereas different CAs provide namespaced isolation for Services without a need for additional filters.

2. The certificate provisioning path can be validated through the flow described in Chapter 4, Section 4.5 of the thesis.
3. High availability of the solution can be implemented through the use of multi-mesh or multi-cluster approaches. Multi-Mesh has its own advantages and disadvantages described in Chapter 6, Section 6.1.
Whenever the question of using this or another approach arises, one should remember that there is no proper and efficient way to handle everything inside one project. Therefore, different solutions and K8s security best practices should be considered to achieve a proper level of security. There are plenty of solutions available today for multi-tenant isolation in K8s, and it is up to organizations to choose what to roll out based on their needs and resources.

Answers for the research questions asked at the beginning of the thesis (described in Chapter 1, Section 1.2) would be:

1. The certificate management process in a service mesh integrated with Istio security mechanisms provides isolation of multiple tenants in a K8s cluster. The implementation of the certificate management system with multiple CAs allows increasing the trust between tenants by distributing reliable certificates for namespaced entities.

2. Istio Authorization Policies are widely used in today’s K8s environment, however, they have plenty of disadvantages, such as Network Policies, described in Chapter 4, Section 4.3 and Section 4.4. The use of multi-CA approach is recommended.

3. The main issue of the solution proposed can be the transparency of the Istio CA key leading to a false sense of security as anyone with administrator rights can read it. Thus, the use of low-level security technologies, such as Intel SGX is recommended. Moreover, security mechanisms should imply integrity, confidentiality, availability, and authenticity. Thus, other layers of security should be considered altogether with the solution.

The thesis conducted an overview of multi-tenancy, K8s, service meshes, and networks. The existing security threats were identified in a multi-tenant
environment in service meshes. The solution was proposed to overcome mentioned challenges. The security and performance experiments were conducted.

The goal of the thesis has been reached as the proposed solution increases the network security in Istio.
Bibliography


[27] Dierks, T., and Rescorla, E. The transport layer security (tls) protocol version 1.2.


Appendix A

Tenant’s Installation

```bash
#!/bin/bash

export server=https://10.237.71.219:6443
export clusterName=istio ---admin@istio

# TENANT ALICE

cat <<<EOF | kubectl apply -f
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: alice
  namespace: bar
rules:
  - apiGroups: [""]
    resources: ["pods", "services"]
    verbs: ["get", "watch", "list"]

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: alice
  namespace: bar
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: alice
subjects:
  - kind: ServiceAccount
    name: alice
    namespace: bar

apiVersion: v1
kind: ServiceAccount
metadata:
  name: alice
  namespace: bar
EOF

kubectl get secrets -n bar
export serviceAccount=alice
export namespace=bar
export secretName=$(kubectl --namespace $namespace get serviceAccount $serviceAccount --o jsonpath='{.secrets[0].name}')
export ca=$(kubectl --namespace $namespace get secret/$secretName --o jsonpath='{.data.ca.crt}')
echo "
apiVersion: v1
kind: Config
clusters:
  - name: $clusterName
    cluster:
      certificate-authority-data: $ca
    server: $server
contexts:
  - name: $serviceAccount$clusterName
    context:
```
APPENDIX A. TENANT’S INSTALLATION

```yaml
cluster: &{.clusterName} namespace: &{namespace} user: &{serviceAccount}
users:
  - name: &{serviceAccount}
    user:
      token: &{token}
current-context: &{serviceAccount}@$[clusterName] > sa-t1.kubeconfig

# TENANT BOB

cat <<EOF | kubectl apply -f
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: bob
namespace: foo
rules:
  - apiGroups: ["", "apps"]
    resources: ["pods", "services", "deployments", "replicasets"]
    verbs: ["get", "watch", "list", "create", "update"

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bob
namespace: foo
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: bob
subjects:
  - kind: ServiceAccount
    name: bob
    namespace: foo

EOF

kubectl get secrets -n foo
export serviceAccount=bob
export namespace=foo
export secretName=$(kubectl -n foo get serviceAccount $serviceAccount -o jsonpath="\{.spec.secretId,\.name\}\")
export ca=$(kubectl -n foo get secret/$secretName -o jsonpath="\{.data.ca.crt\}" | base64 -d)

echo "Roles and service accounts in bar namespace"
kubectl get roles -n bar
kubectl get serviceaccounts -n bar
echo "Roles and service accounts in foo namespace"
kubectl get roles -n foo
kubectl get serviceaccounts -n foo
```
Appendix B

Multi-CA Installation

CERT-MANAGER INSTALLATION

```
kubectl apply -f multi-ca/cert-manager.yaml
```

CREATION OF TENANT’S SECRETS

```
kubectl create secret generic foo-ca -n cert-manager \
    --from-file=foo/tls.crt \
    --from-file=foo/tls.key
```

```
kubectl create secret generic bar-ca -n cert-manager \
    --from-file=bar/tls.crt \
    --from-file=bar/tls.key
```

CREATION OF CA ISSUERS

```
cat ca-issuer.yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: foo
spec:
  ca:
    secretName: foo-ca

apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: bar
spec:
  ca:
    secretName: bar-ca
```

```
cat istio-issuer.yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: istio-system
spec:
  ca:
    secretName: bar-ca
```

ISTIO INSTALLATION WITH CERT-SINGER-DOMAIN

```
istioctl install --set values.global.logging.level=debug -f istio-certs.yaml
```

```
cat istio-certs.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    pilot:
      kbs:
        env:
          - name: CERT-SIGNER-DOMAIN
            value: clusterissuers.cert-manager.io
          - name: EXTERNAL-CA
            value: ISTIODRA-KUBERNETES-API
        overlays:
          certificate signing by custom signer
          - kind: ClusterRole
```

72
APPENDIX B. MULTI-CA INSTALLATION

```yaml
name: istiod--clusterrole--istio--system
patches:
  - path: rules[−1]
    value: |
      apiGroups:
        - certificates.k8s.io
        - clusterissuers.cert-manager.io/*
      resources:
        - signers
      verbs: approve
    values:
      pilot:
        image: pilot--multica:latest
      proxy:
        image: proxyv2--multica:latest

DEPLOYMENT OF TENANT'S WORKLOADS

kubectl create ns foo
kubectl apply -f <(istioctl kube-inject -f samples/httpbin/httpbin.yaml) -n foo
kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.yaml) -n foo

kubectl create ns bar
kubectl apply -f <(istioctl kube-inject -f samples/httpbin/httpbin.yaml) -n bar
kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.yaml) -n bar
```