Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Aleksandr Kurbatov

Design and implementation of secure communication between microservices

Master's Thesis
Espoo, December 31, 2020

Supervisor: Professor Tuomas Aura
Advisor: Antti Jaakkola, M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Aleksandr Kurbatov

Title:
Design and implementation of secure communication between microservices

Date:	December 31, 2020	**Pages:**	50
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Tuomas Aura		
Advisor:	Antti Jaakkola, M.Sc. (Tech.)		

The demand for the microservice architecture is rapidly growing by every day, and so is the problem of its security. The objective of the study was to analyze threats against the Kubernetes threats and find a solution for how to address these threats without degrading system performance and overcomplicating the deployment process.

The thesis concentrates on the following researching questions: (1) What are the solutions that exist for? (2) What are the requirements for the solution design? (3) How to develop and evaluate such a method?

The study reviews the multiple available service mesh solutions and chooses to utilize the Istio product for the solution implementation. Based on the composite threat model, we defined a set of requirements for the solution design. The designed method constitutes a script for the automated deployment of Istio service mesh and secure connection establishment.

From the acquired results it can be concluded that the designed method has been able to prevent and mitigate most of the discovered threats and comply with the security and performance requirements.

Keywords: cloud computing, Kubernetes, Istio, Linkerd, Consul, service mesh, network security

Language: English

Acknowledgements

I would like to express my gratitude to my thesis supervisor Tuomas Aura and Alisa Vorobeeva for directing the research and giving helpful recommendations. I also want to thank my advisor Antti Jaakkola and Adam Peltoniemi for helping both on the technical and formal sides of the thesis. Special thanks to Lauri Salmio for being so attentive and respectful to me during my internship at Ericsson.

In addition, I want to thank my family and friends who have been supporting me throughout my Aalto studies even being in a thousands of kilometers away from me.

Espoo, December 31, 2020

Aleksandr Kurbatov

Contents

Chapter 1

Introduction

Recently, cloud computing has become an essential component in the functioning of any large business. Thousands of companies lease and rent cloud servers to efficiently process user data and provide their services to millions of users. However, complex cloud products often require precise communication between a large number of microservices. Container orchestrators, such as Kubernetes, effectively addresses this problem. Kubernetes runs and coordinates containerized applications across a cluster of machines. It is a platform designed to completely manage the life cycle of containerized applications and services and successfully build and deploy reliable, scalable, and distributed systems [20]. Nevertheless, with new technology coming, vulnerabilities also appear.

At this moment, the data life circle on the internet and local networks is sufficiently secured. Right now we have HTTPS and TLS protocols, Virtual Private Networks, authentication protocols as OAuth2 and SAML, and much more for secure communication. However, Kubernetes focusing more on workload orchestration and does not support most of the security measures for communication between microservices that are working inside of Kubernetes environment. Developers that use the Kubernetes environment for their business applications and want to implement more advanced security measures will have to do it by themselves. In order to do this, developers should manually configure the whole system and, probably, interfere into the code of most microservices. As the result, developers could spend a significant amount of time finding the proper solution. Moreover, the scalability of such a solution may not be guaranteed.

To address the problem of the security of microservices communication, a service mesh can be used. A service mesh can be described as an infrastructure layer that is responsible for the communication between services. It deploys special sidecar proxies throughout the working environment that

intercepts all network communication between microservices. The service mesh configures and manages the communication, including traffic control, certificate management, and access control policy configuration.

Kubernetes environment and service mesh are still new and developing technologies, the security issues of which are relevant and demand more research. Thus, it is a relevant and challenging task to explore the usage of the service mesh technology in the Kubernetes environment in order to improve its security state and decrease the number of possible threats.

The goal of this thesis is to decrease the number of threats, do not significantly degrade the system's performance, and substantially simplify the deployment process. This thesis will consider and analyze the Istio service mesh.

The tasks of the thesis are:

a. Review and analysis of literary sources and existing solutions in the field of existing options for ensuring the safety of microservice communication.
b. Development of requirements for implementation in a corporate system of a method for ensuring the safety of microservice communications.
c. Development of a method for securing microservice communications within a Kubernetes virtual environment.
d. Conduct experimental studies to evaluate the effectiveness of the developed method.

This thesis focuses on communication security between microservices in a service mesh. It will try to present how a service mesh can improve the security of the Kubernetes environment, how to deploy service mesh and answer if it is a sufficient enough solution. Service mesh has a lot of other functions and opportunities, but they are not relevant to this thesis because we are concentrating on the security point of view.

The rest of the thesis is structured as follows: Chapter 2 presents the concepts of Kubernetes, microservices, service meshes, and possible attacks on microservices. Chapter 3 introduces a threat model. Chapter 4 describes the requirements for the solution. Chapter 5 illustrates the system's architecture design and describes the method design details, while chapter 6 shows the evaluation of the proposed solution. Finally, the thesis is concluded with chapter 7.

Chapter 2

Background

2.1 Cloud computing

Cloud computing already became a commonness and almost every business in the information technology sphere faced or works with it. Moreover, almost everyone has used at least once the products based on a cloud computing model, like Google Docs, Netflix, or Spotify [31]. The definition of cloud computing term was officially introduced by the National Institute of Standards and Technology (NIST) in 2011 [28]. It refers to cloud computing as a model that provides on-demand, ubiquitous, and available access to different manageable computer resources with the expeditious provision and effortless management.

The utilization of the cloud computing model brought a lot of benefits. For example, it decreased cost and energy-saving of servers maintaining, made the deployment of applications scalable and agile, improved the efficiency of resource management, and much more.

Cloud computing has different service delivery models. The services, which are represented in Figure 2.1, are classified into three categories: software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS) [9].

Software as a service model allows a provider to manage and supply applications and services for outer clients and developers. These applications can be utilized through web services or web browsers. The client does not need to know about the application structure and its management. To maintain Saas, service level agreements are used. For example, Google Maps is the application provided by the Saas model. Summarizing, the Saas model provides server mobility, service migration, and hosting, so a user could effortlessly utilize an application.

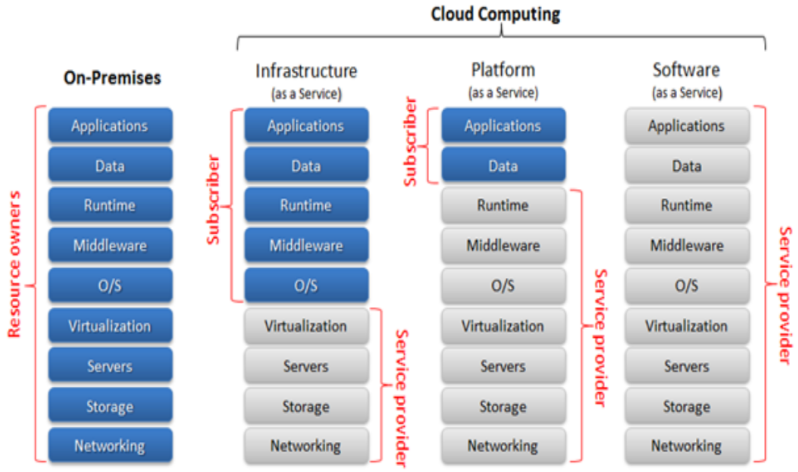


Figure 2.1: Service delivery model (source: `http://mural.maynoothuniversity.ie/2970/1/GC_Intro_Cloud.pdf`)

Platform as a service allows users to implement their applications on the provider cloud infrastructure. Instead of supplying services, the providers supply libraries and tools in order to deploy the application itself. Applications are executed on the virtual platform in a transparent manner, similar to the Google Apps Engine and Morph Labs. The platform layer provides value-added services from both a technical and a business perspective, for example, Salesforce and xRM enables the whole application life cycle for the development, deployment, and management of tailored business applications.

Infrastructure as a service model provides data store, networking, and computing resources as services and allows to users manage the virtual infrastructure. The users are served with the provider's resources using a multi-tenant model. This model helps to dynamically manage virtual and physical resources as well as assign and reassign them for client's needs. As an example of the PaaS model is Google Bigtable and Amazon Dynamo. OpenFlow and Google FS are also offering IaaS, however, on a smaller scale [9].

2.2 Docker and Kubernetes

Docker is an open-source platform that provides developing, shipping, and running application by providing software virtualization with a namespace, filesystem, and processes isolation on the same operating system of the bare-metal or virtual machine. Docker helps effortlessly and reliably manage con-

tainerized applications. This also led to the birth of smaller services being packed as independent software units [24].

Kubernetes is an open-source platform for containerized applications management within PaaS cloud models. Kubernetes is frequently called a container orchestration platform that helps to optimize the process of managing, scaling, and deploying containerized applications. The Kubernetes management system is realized through "pods" â groups of containers. Pods could be dynamically scaled depending on the amount of workload.

With significant interest in supporting cloud native applications (CNA), Kubernetes provides a useful approach to achieve this. One of the key requirements for CNA is support for scalability and resilience of the deployed application, making more effective use of on-demand provisioning and elasticity of cloud platforms. Containers provide the most appropriate mechanism for CNA, enabling rapid spawning and termination compared to Virtual Machines (VMs). The process management origin of container-based systems also aligns more [27].

2.3 Service Mesh

A service mesh can be described as a specific infrastructure layer that manages communication between the services of a containerized system via load balancing, routing, authentication, encryption, and monitoring. Service mesh allows a developer to configure the network conduct, traffic flow, and node identities through policy enforcement.

It can be looked upon as a networking model that sits at a layer of abstraction above the transport layer of the Open System Interconnection (OSI) model (e.g., Transport Control Protocol/Internet Protocol (TCP/IP)) and addresses the service's session layer (Layer 5 of the OSI model) concerns. However, a fine-grained authorization may still need to be performed at the microservice since that is the only entity that has the full knowledge of the business logic. A service mesh conceptually has two modulesâthe data plane and the control plane. The data plane carries the application request traffic between service instances through service-specific proxies. The control plane configures the data plane, provides a point of aggregation for telemetry, and provides APIs for modifying the behavior of the network through various features, such as load balancing, circuit breaking, or rate-limiting. Service meshes create a small proxy server instance for each service within a microservices application. This specialized proxy car is sometimes called a "sidecar proxy" in service mesh parlance. The sidecar proxy forms the data plane, while the runtime operations needed for enforcing security (ac-

cess control, communication-related) are enabled by injecting policies (e.g., access control policies) into the sidecar proxy from the control plane. This also provides the flexibility to dynamically change policies without modifying the microservices code [10].

2.3.1 Istio

Istio is an open-source service mesh solution that manages the communication between microservices. Istio can be effortlessly integrated due to its integral API. The Istio project was build by Google and IBM with using of Envoy technologies of Lyft. Istio is used as a default service mesh solution by companies like Microsoft, Google, and IBM. Istio is a Kubernetes-based solution and does not support other platforms.

Istio was the first service mesh that implemented the support of additional features, such as deep-dive analytics.

Istio separates the data management on data and control planes. It is achieved via the implementation of proxy sidecars which allows to cache data, preventing the back data flow to the control plane for every call [24]. The control planes are pods that also run in the Kubernetes cluster, allowing for better resilience in the event that there is a failure of a single pod in any part of the service mesh.

2.3.2 Linkerd

Linkerd is arguably the second most popular service mesh on Kubernetes and, due to its rewrite in v2, its architecture mirrors Istio's closely, with an initial focus on simplicity instead of flexibility [24]. This fact, along with it being a Kubernetes-only solution, results in fewer moving pieces, which means that Linkerd has less complexity overall. While Linkerd v1.x is still supported, and it supports more container platforms than Kubernetes; new features (like blue/green deployments) are focused on v2. primarily.

Linkerd is unique in that it is part of the Cloud Native Foundation (CNCF), which is the organization responsible for Kubernetes. No other service mesh is backed by an independent foundation.

2.3.3 Consul

Consul is a full-feature service management framework, and the addition of Connect in v1.2 gives it service discovery capabilities which make it a full Service Mesh. Consul is part of HashiCorp's suite of infrastructure management products; it started as a way to manage services running on Nomad and

Properties	Istio	Linkerd	Consul
mTLS	Yes	Yes	Yes
Certificate Management	Yes	Yes	Yes
Authentication and Authorization	Yes	Yes	Yes
TCP	Yes	Yes	Yes
Traffic Rate Limiting	Yes	No	Yes
Traffic testing	Yes	Limited	No
Monitoring	Yes, with Prometheus	Yes, with Prometheus	Yes, with Prometheus
Distributed Tracing	Yes	Some	Yes
Multicluster support	Yes	No	Yes
Installation	Helm and Operator	Helm	Helm

Table 2.1: Service Mesh Comparison

has grown to support multiple other data center and container management platforms including Kubernetes.

Consul Connect uses an agent installed on every node as a DaemonSet which communicates with the Envoy sidecar proxies that handle routing forwarding of traffic.

2.3.4 Service Mesh comparison

Finally, after a brief description of each service mesh technology, we will analyze the comparison of service mesh characteristics. The results of the comparison are represented by Table 2.1.

As we can see, the Istio is the most effective service mesh at the moment. Therefore, it was decided to utilize the Istio service mesh for our solution. However, currently, Istio technology is quite complex and not friendly to developers who just started to work with it. To overcome this problem, our solution will include the deployment script, allowing us to automate and facilitate the installation and usage of Istio service mesh.

2.4 Network security

Transport Layer Security (TLS) is a network security protocol that provides privacy and data integrity during network communication. The protocol includes the Record and Handshake layer [14].

The TLS Record Protocol provides connection security that has two basic properties:

- The privacy of the connection. It is guaranteed by using symmetric encryption protocols, such as AES, RC4, and more. For each connection, the protocol uniquely generates the keys using the secret provided by the TLS Handshake protocol. However, the Record protocol can be also used without any encryption.

- The reliability of the connection. It is achieving through a message integrity check using the message authentication code. A message authentication code (MAC) is the code that helps to confirm the authenticity and integrity of a whole message. MAC is using a secure hash function, such as SHA-1, and a key-value to encrypt the user's identity information. To verify the massage's authenticity and integrity, the recipient also generates the MAC and compares them.

The mTLS protocol is an optional modification for regular TLS protocol that allows authenticating two parties instead of one at the same time.

By default, the TLS protocol only proves the identity of the server to the client using X.509 certificate, and the authentication of the client to the server is left to the application layer [14]. TLS also offers client-to-server authentication using client-side X.509 authentication. As it requires provisioning of the certificates to the clients and involves a less user-friendly experience, it's rarely used in end-user applications.

Mutual TLS authentication (mTLS) is much more widespread in business-to-business (B2B) applications, where a limited number of programmatic and homogeneous clients are connecting to specific web services, the operational burden is limited, and security requirements are usually much higher as compared to consumer environments.

Most Mutual authentication is machine-to-machine, leaving it up to chance whether users will notice (or care) when the remote authentication fails (e.g. a red address bar browser padlock, or a wrong domain name). Non-technical mutual-authentication also exists to mitigate this problem, requiring users to complete a challenge, effectively forcing them to notice, and blocking them from authenticating with false endpoints.

Chapter 3

System architecture

Before we start our solution design, we need to define the environment and application where it will be deployed. This section represents host machine characteristics, Kubernetes environment, and integrative application details.

First, our solution will be deployed on the **virtual machine** (VM) with **Ubuntu 16.04** operating system. The VM has the following characteristics:

- 20 Gb of free disk space
- RAM: 16 Gb
- Processor frequency: 2.0 GHz
- 8 CPUs

To simulate the Kubernetes environment locally we will utilize the **Minikube v.1.9.2.** Minikube is an open-source software solution that helps efficiently and simply simulate a local Kubernetes cluster on any operating system. Due to Minikube already being installed on the VM, it does not require the deploying of a cluster in any container or virtual machine manager.

To integrate and test our solution, we also need to deploy some applications on top of our cluster. To keep the application environment simple for the thesis work, we will utilize a demo application. Therefore, it was decided to implement the Bookinfo application.

The Bookinfo is a simple application intended for keeping and displaying the information about a book, such as description, number of pages, ISBN, and its reviews [2].

The Bookinfo application consists of 4 different microservices:

- **Productpage**: The **productpage** service requests the reviews and details services for page population

- **Details**: The **Details** microservice keeps information about the books.
- **Reviews**: The **reviews** service stores the book reviews and requests the **ratings** service.
- **Ratings**: The **ratings** service stores the book rating score that also showed in a book review.

The **review** service has three following versions:

- First version (reviews-v1): the service version which does not **requests** the **ratings** service.
- Second version (reviews-v2): shows the ratings as 1 to 5 black stars and requests the ratings service.
- Third version (reviews-v3): shows the ratings as 1 to 5 red stars and requests the **ratings** service.

The Figure 3.1 shows the end-to-end architecture of the Bookinfo application.

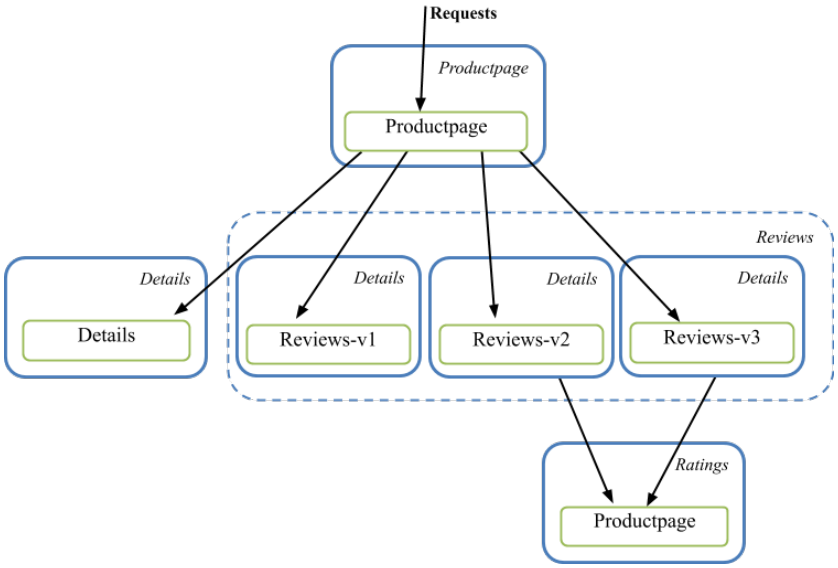


Figure 3.1: Bookinfo Application architecture

The Bookinfo application contains microservices that are written in different programming languages. This is presenting the interesting service mesh

example, particularly because of the multitude of services, languages, and versions for the reviews service.

For collecting and analyzing of the system information was installed **Prometheus v.2.8.** Prometheus is a standalone open-source project, designed for system monitoring and alerting. Prometheus allows collecting all required information about cluster components and the Kubernetes system in general. We utilize Prometheus to get information about such system parameters as CPU usage, memory usage, and filesystem usage.

To visualize the gathered by Prometheus data, we also used the **Grafana v.7.0** application. Grafana is an open-source project, designed to visualize, alert, and query on metrics and logs of the system. Grafana is perfectly compatible with Kubernetes and Prometheus, which allows effectively and effortlessly to collect the data and represent the state of the system.

Chapter 4

Requirements

Before we design the solution, we should thoroughly analyze the threats of the system and describe the necessary requirements that the solution should suffice. At first, we will formulate the attacker model. Secondly, based on the attacker model the threat model will be developed. Finally, the section will present the list of requirements for the solution design.

4.1 Attacker model

The attacker model defines malicious actors and helps to determine which protections, if any, are necessary to mitigate or remediate a threat. The actors of the attacker model will be used in the threat model. The attacker model also will describe actors of the system, who may be impacted by, or enticed to undertake an attack. The devised attacker model is based on the Threat Actors model from the Kubernetes Threat model study made by Cloud Native Computing Foundation in 2019 [17].

The Cloud Native Computing Foundation (CNCF) is a rapidly growing organization that is interested in fostering and building sustainable environments such as a public, private and hybrid cloud. Currently, CNCF comprises 150,000 members, includes over 140 top companies and startups, including Microsoft, Apple, Amazon, and graduates open-source projects like Kubernetes, Helm, Envoy, and much more. In 2019, the CNCF published a major study on Kubernetes security [1]. This study includes the Kubernetes Security Review, Attacking and Defending Kubernetes Installations, Whitepaper, and Threat Model. As the basis of our attacker model, we consider the Threat Actors model of the Kubernetes Threat model [17]. The attacker model is presented in Table 4.1.

Attacker	Description
Malicious Internal User	A user, such as an administrator or developer, who uses their privileged position maliciously against the system, or stolen credentials used for the same.
Internal actor	An attacker who had transited one or more trust boundaries, such as an attacker with container access.
External actor	An attacker who is external to the cluster and is authenticated.
Administrator	An actual administrator of the system, tasked with operating and maintaining the cluster as a whole.
Developer	An application developer who is deploying an application to a cluster, either directly or via another user (such as an Administrator).
End User	An external user of an application hosted by a cluster.

Table 4.1: Attacker model

4.2 Threat model

The formulation of threat modeling is an important and crucial part of this thesis. The threat model defines the list of the system's relevant threats and vulnerabilities. Usually, the creation of the threat model is based on a methodology. There are numerous threat modeling methodologies that can be implemented such as STRIDE, P.A.S.T.A, Trike, VAST, etc. Despite a large number of existing methodologies, just a small part of them can be implemented for Kubernetes threat modeling.

The development of our threat model was based on the CNCF Kubernetes Threat model [17] and Microsoft's Threat matrix for Kubernetes [37].

In April 2020, Yossi Weizman, the Security Research Software Engineer in Azure Security Center, published his study on Kubernetes threat modeling [37]. The model is based on the MITRE ATTCK framework [11] and includes numerous Kubernetes threats and vulnerabilities, which in the paper are represented as tactics and techniques respectively.

Based on the above threat modeling methodologies, we developed the threat model in the scope of the security of Kubernetes microservices communication, which is represented in Table 4.2.

	Threat	Attacker	Description
1	Man-in-the-Middle attack between cluster components	Malicious Internal User, Internal actor, or External actor that got privileges of an Internal actor	By default, non-system components have unverified connections, which make them vulnerable for Man-in-the-Middle attacks
2	Sniffing of internal traffic	Malicious Internal User, Internal actor, or External actor that got privileges of an Internal actor	Attacker can intercept the internal traffic using sniffing programs and get the sensitive information from different microservices
3	Unauthorized connection to another pods and applications API	External actor, Internal actor	If the attacker could compromise a cluster's container, he will be able to gain access to other containers, since the default Kubernetes network setting allows to reach any other container
4	Exploiting the access to Kubelet API	Malicious Internal User, Internal actor, or External actor that got privileges of an Internal actor	Kubelet is the node agent that is executed on each node to control its pods efficiency. Kubelet utilizes the read-only API service and does not require authentication of TCP port (10255). If the attacker has a compromised host, he can request Kubelet API to acquire the information about running pods or node parameters, like memory and CPU usage.
5	Network scanning	External actor	Attackers may scan the Kubernetes network for cluster data, such as running workloads, or known vulnerabilities. Since, initially Kubernetes does not restrict the services communication, attackers with compromised container can try to probe the network
6	Changing the VM settings through compromised container with root privileges	Malicious Internal User, Internal actor, or External actor that got privileges of an Internal actor	If the attacker could get access to a root role inside of compromised container, he will be able to change VM settings, using OS libraries or procfiles

Table 4.2: Threat model

4.3 Secure communication

Concluding from the devised threat model above, we infer secure communication is still a very challenging problem. In environments like Kubernetes commonly use TLS protocol for communication between services. The TLS protocol allows confirmation of one side of the authentication process. Usually, it is the client-side.

However, when it comes to communicating between microservices, it is very important to authenticate both sides to avoid attacks like Man-in-the-Middle. In this case, the mutual TLS protocol can be implemented. Mutual TLS is the variation of the regular TLS protocol, while it also can authenticate both sides during the authorization process. It is a very useful and effective method to secure communications in the Kubernetes network. Moreover, mutual TLS also provides transparent encryption and applies certificates for signatures.

Thus, we set a requirement that secure communications between microservices in Kubernetes should be implemented via the mutual Transport Layer Security (mTLS) protocol. In order to properly comply with this requirement, we should set a couple of other requirements, like certificate management and encryption provision.

4.4 Certificate life-cycle management

In order to properly maintain mTLS protocol, the solution should implement a reliable and flexible certificate management system. It should operate with X.509 certificates. X.509 is a well-known standard that specifies the format of public-key certificates. It is commonly used in TLS and SSL protocols, which allows the implementation of HTTPS for secure web browsing.

Such a certificate system should be able to create certificates, sign them with its own Certificate Authority, distribute them between all services, and maintain their life cycle (regular checking, revoking, recreating, etc.). Thus, we set a requirement for our solution to provide a robust certificate management system.

4.5 Encryption

In order to protect the sensitive traffic from disclosure by the third party, the circulating in system information should be encrypted. In the case of communication through TLS, the data is encrypted symmetrically. It is

achieved through generating keys uniquely for each connection and using the shared secret, that was set during the TLS handshake. The secret sharing is secured from eavesdropping and modifications from external attackers.

Thus, we set the requirement for the designed solution that has to implement reliable transparent TLS encryption for securing of traffic between microservices.

4.6 Automation

As we mentioned in the previous section, some technologies, like Istio, could be quite complex for implementation. Istio is a relatively new product, trying to provide more secure and advanced service mesh technology. However, it causes the complexity of managing and configuring such a system, thus confusing inexperienced developers.

In that case, we should specify the following requirement: the solution should be automated for deployment and not cause any problems during its installation.

4.7 Scalability

Kubernetes system is complex and dynamic. It allows to effortlessly control the load balancing and number of workloads. The property of a system, that allows controlling the number of used resources in accordance with the increasing workload is called scalability. Scalability is a very important property of the Kubernetes environment. It always maintains the required number of pods for continuous service execution and proper allocation of the working load. The designed solution should fit into a dynamic and scalable system of Kubernetes and provide secure communication to any number of services.

Thus, we set the requirement, that the designed solution should be scalable and consistently support any number of working pods, without any interception into the application's code.

4.8 Performance

Kubernetes is a broad and complex environment that can adapt to a high amount of work and control the workloads over all services accordingly. Deployment of new technology, such as service mesh, into the Kubernetes environment, will affect the whole system. Thus, it is vital to ensure that additional solutions will not degrade the system's performance. Thus, the

designed solution should not degrade the performance of the system and do not conflict with any components of the basic environment.

Chapter 5

Implementation

5.1 Solution architecture

Based on the previously discussed system's characteristics and requirements, the Istio service mesh was chosen for our solution design.

Istio service mesh allows the facilitation and organization of a set of deployed services with monitoring, load balancing, service-to-service communication, and more, without any changing of application code. Istio creates proxy sidecars and deploys them into services all over the cluster. These sidecars in turn intercept all traffic between containers. In order to control and manage the system, Istio fulfills the following functionalities:

- Management of retries, fault injections, failovers and routing for traffic behavior control.
- Robust authentication and authorization of microservices based on the identity concept to provide protected communication between them.
- Monitoring of traffic, metrics, and logs all over the cluster.
- Optimized load balancing of different types of traffic, such as HTTP, gRPC, WebSocket, and TCP.
- Evaluation of quotas and system limits by additional API configuration and policy layer.

Figure 5.1 shows the simplified structure of how Istio works. Istio supports the system's extensibility of deployment demands by intercepting and configuring mesh traffic.

In terms of security, Istio maintains encryption, authentication, and authorization of service communication and establishes the underlying safe

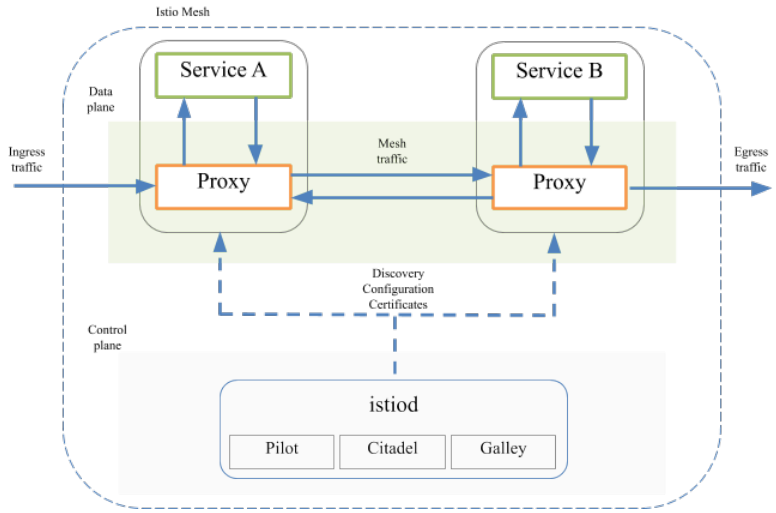


Figure 5.1: Istio architecture

communication channel. To provide these and more security features, Istio implements the following elements:

- Envoy proxies for telemetry management and auditing
- The configuration API server distributes to the proxies:
  - secure naming information
  - authentication policies
  - authorization policies
- A Certificate Authority (CA) for key and certificate management
- Sidecar and perimeter proxies work as Policy Enforcement Points (PEPs) to secure communication between clients and servers.

Before we proceed to the description of related security details, we need to introduce the concept of identity in Istio's service mesh. Before any external and internal services' communication, Istio requests identities of both sides to establish authentication. On the client-side, Istio is checking the information about the client's secure naming to check for its authorization. On the server-side, Istio checks authorization policies, analyzes the access journal, monitors what workloads the clients are using, and denies the client's access if the payment for access to the workloads fails. In other words, identity is Istio's

model for the determination of the request's origin. It represents a human user, an individual workload, or a group of workloads.

Istio provides X.509 certificates for every workload identity. Inside every Istio proxy sidecar, the Istio agents communicate with istiod, one of the core Istio's components, to distribute certificates and keys for all over the mesh. Figure 5.2 represents the scheme of certificate distribution by Istio service mesh.

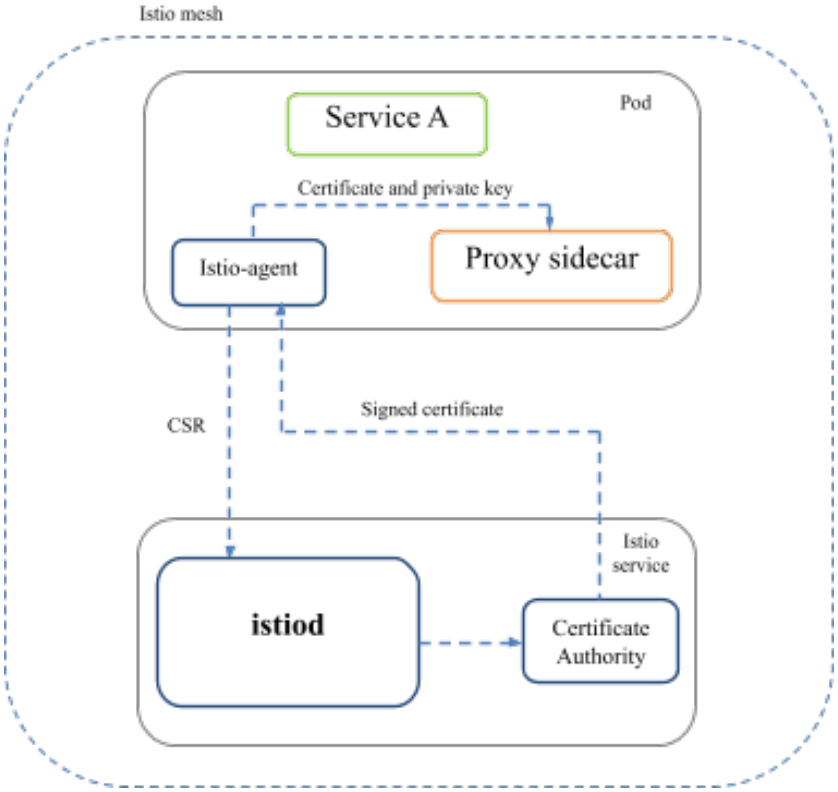


Figure 5.2: Certificate distribution scheme

The simplified key and certificate distribution process works as follows. Initially, the service's Istio agent sends to istiod's certification authority (CA) the proxy's certificate and key through the certificate signing request (CSR). Secondly, the CA signs the received CSR and generates the new certificate, and sends it back to the Istio agent. Finally, the Istio agent sends the new certificate and private key back to the Envoy proxy sidecar. The outlined above process is continuously repeating for key and certificate rotation for all services.

Generally, Istio provides two types of authentication: the peer authentication service-to-service authentication; and request authentication for end-user authentication. Since our goal is to provide secure communication between services, we will investigate only peer authentication.

The peer authentication implies the realization of mutual Transport Layer Security protocol (mTLS) for services authentication. It also includes the provision of strong identities, key and certificate management systems for each service, and secure service-to-service communication. The latter function is maintained through Envoy proxies. Every time when a service receives or sends traffic, it is intercepted by the Envoy proxy sidecar. The proxy sidecar initializes the mTLS handshake, during which it verifies the requested service permissions in the server certificate. If the handshake succeeds, the traffic becomes encrypted with the transparent TLS encryption protocol and Istio continues the mTLS connection by forwarding the traffic between proxy sidecars.

The peer authentication is controlled with self-titled policies. The peer authentication policies define the mTLS rules that Istio compels to workloads. The policies support three modes:

- PERMISSIVE: the policy allows the receiving of regular text traffic, as well as mTLS traffic.
- STRICT: policy allows to receive only mTLS traffic.
- DISABLE: policy disables mTLS on the workload.

The permissive mode is set on all Envoy sidecars by default.

The automation part of our solution is represented by a script, which allows automating the service mesh installation, the injection of the proxy sidecars, and the creation of authentication policies. The implemented code is made in the regular Linux bash scripting language and is entirely presented in Appendix A. The devised code includes three major functions:

1. The automated installation. The *install* command launches the sequence of commands for downloading and setting the required software, libraries, and variables, installing the Istio service mesh into the system, and preparing it for subsequent work.
2. The automated injection. The *inject* command is responsible for the integration of Istio's proxy sidecars into the specified services. It is a vital part of establishing secure communication between microservices. Before Istio can start to work with the traffic, the proxy sidecars

should be deployed into each pod that needs to be secure. The process of such deployment is called injection. Only after the injection is finished and sidecar proxies are deployed into all required pods, Istio can finally initiate the interception and securing of incoming and outcoming service traffic. After the input of the *inject* command, the user should specify the particular Kubernetes namespace. It allows injecting the Istio sidecars into every pod of the specified namespace. The user can also specify the name of the specific deployment after specifying the namespace, which allows integrating the proxy sidecar not in the whole namespace but only in the specific deployment of the specified namespace. If the user did not specify any namespace, the script will inject the *default* namespace. After the injection is finished, all involved pods will be restarted in order to launch the proxy sidecars.

3. The automated enabling of mTLS. After all required pods are injected with the Envoy proxy sidecars, the user can initialize the setup of authentication rules of injected pods to enable the mTLS communication. The *secure* command creates the Peer Authentication policy for specified pods or namespaces in order to establish the type of secure mTLS connection for them. Users are able to choose if they want to secure a namespace or the whole Kubernetes system. To achieve this, the user should enter the namespace name or the *system* command respectively after having entered the *secure* command. As in the case of the *inject* command, users can specify the pod name after the name of the namespace, to secure the specified pod.

5.2 Deployment

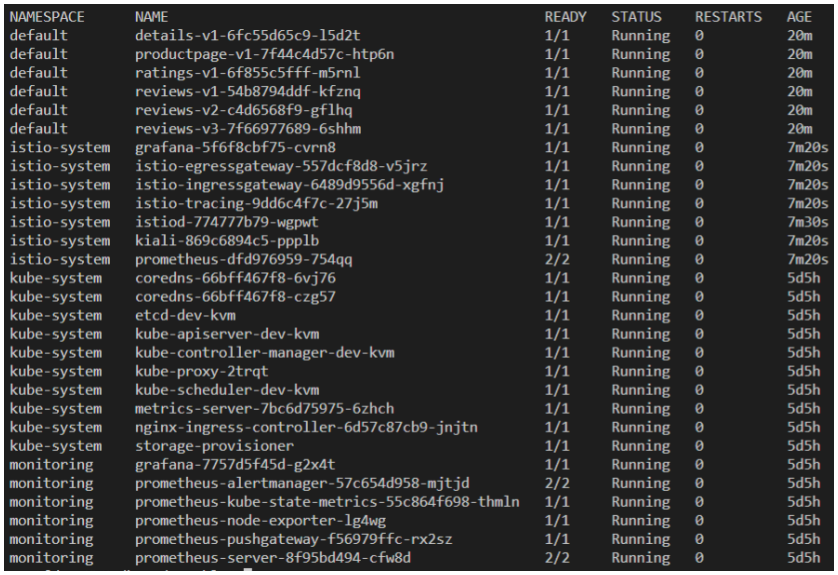
As we mentioned in chapter 3 3, we simulate our Kubernetes environment via Minikube, which already has installed Prometheus, Grafana, and the Bookinfo application. First of all, we will check if all components of our system work correctly. To ensure this, we will launch the following command: `kubectl get pods -all-namespaces`. Thus, we will see the status of the application and monitoring system pods. Figure 5.3 shows the result of the command.

```
NAMESPACE     NAME                                          READY   STATUS    RESTARTS   AGE
default       details-v1-6fc55d65c9-l5d2t                   1/1     Running   0          87s
default       productpage-v1-7f44c4d57c-htp6n               1/1     Running   0          87s
default       ratings-v1-6f855c5fff-m5rnl                   1/1     Running   0          87s
default       reviews-v1-54b8794ddf-kfznq                   1/1     Running   0          87s
default       reviews-v2-c4d6568f9-gflhq                    1/1     Running   0          87s
default       reviews-v3-7f66977689-6shhm                   1/1     Running   0          87s
kube-system   coredns-66bff467f8-6vj76                      1/1     Running   0          5d5h
kube-system   coredns-66bff467f8-czg57                      1/1     Running   0          5d5h
kube-system   etcd-dev-kvm                                  1/1     Running   0          5d5h
kube-system   kube-apiserver-dev-kvm                        1/1     Running   0          5d5h
kube-system   kube-controller-manager-dev-kvm               1/1     Running   0          5d5h
kube-system   kube-proxy-2trqt                              1/1     Running   0          5d5h
kube-system   kube-scheduler-dev-kvm                        1/1     Running   0          5d5h
kube-system   metrics-server-7bc6d75975-6zhch               1/1     Running   0          5d5h
kube-system   nginx-ingress-controller-6d57c87cb9-jnjtn     1/1     Running   0          5d4h
kube-system   storage-provisioner                           1/1     Running   0          5d5h
monitoring    grafana-7757d5f45d-g2x4t                      1/1     Running   0          5d5h
monitoring    prometheus-alertmanager-57c654d958-mjtjd      2/2     Running   0          5d5h
monitoring    prometheus-kube-state-metrics-55c864f698-thmln 1/1    Running   0          5d5h
monitoring    prometheus-node-exporter-lg4wg                1/1     Running   0          5d5h
monitoring    prometheus-pushgateway-f56979ffc-rx2sz        1/1     Running   0          5d5h
monitoring    prometheus-server-8f95bd494-cfw8d             2/2     Running   0          5d5h
```

Figure 5.3: The initially running pods

We can see that all components of Kubernetes, Bookinfo, Prometheus, and Grafana are running and working properly.

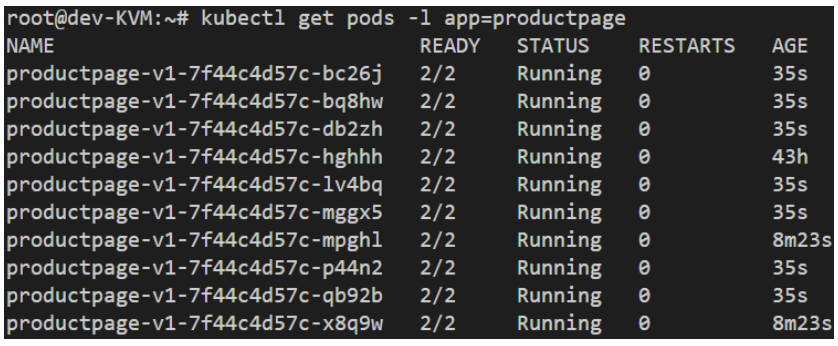
Now, to deploy and set our solution, we will use our designed script called `auto-istio.sh`. First, to install Istio service mesh into our system, we will use the `install` command of the script. So, the command will look like this: `./auto-istio.sh install`. After the installation process is completed, we should check if the service mesh components are working properly. Once again, we will use the `kubectl get pods -all-namespaces` command. The result can be seen in Figure 5.4.



NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	details-v1-6fc55d65c9-15d2t	1/1	Running	0	20m
default	productpage-v1-7f44c4d57c-htp6n	1/1	Running	0	20m
default	ratings-v1-6f855c5fff-m5rnl	1/1	Running	0	20m
default	reviews-v1-54b8794ddf-kfznq	1/1	Running	0	20m
default	reviews-v2-c4d6568f9-gflhq	1/1	Running	0	20m
default	reviews-v3-7f66977689-6shhm	1/1	Running	0	20m
istio-system	grafana-5f6f8cbf75-cvrn8	1/1	Running	0	7m20s
istio-system	istio-egressgateway-557dcf8d8-v5jrz	1/1	Running	0	7m20s
istio-system	istio-ingressgateway-6489d9556d-xgfnj	1/1	Running	0	7m20s
istio-system	istio-tracing-9dd6c4f7c-27j5m	1/1	Running	0	7m20s
istio-system	istiod-774777b79-wgpwt	1/1	Running	0	7m30s
istio-system	kiali-869c6894c5-ppplb	1/1	Running	0	7m20s
istio-system	prometheus-dfd976959-754qq	2/2	Running	0	7m20s
kube-system	coredns-66bff467f8-6vj76	1/1	Running	0	5d5h
kube-system	coredns-66bff467f8-czg57	1/1	Running	0	5d5h
kube-system	etcd-dev-kvm	1/1	Running	0	5d5h
kube-system	kube-apiserver-dev-kvm	1/1	Running	0	5d5h
kube-system	kube-controller-manager-dev-kvm	1/1	Running	0	5d5h
kube-system	kube-proxy-2trqt	1/1	Running	0	5d5h
kube-system	kube-scheduler-dev-kvm	1/1	Running	0	5d5h
kube-system	metrics-server-7bc6d75975-6zhch	1/1	Running	0	5d5h
kube-system	nginx-ingress-controller-6d57c87cb9-jnjtn	1/1	Running	0	5d5h
kube-system	storage-provisioner	1/1	Running	0	5d5h
monitoring	grafana-7757d5f45d-g2x4t	1/1	Running	0	5d5h
monitoring	prometheus-alertmanager-57c654d958-mjtjd	2/2	Running	0	5d5h
monitoring	prometheus-kube-state-metrics-55c864f698-thmln	1/1	Running	0	5d5h
monitoring	prometheus-node-exporter-lg4wg	1/1	Running	0	5d5h
monitoring	prometheus-pushgateway-f56979ffc-rx2sz	1/1	Running	0	5d5h
monitoring	prometheus-server-8f95bd494-cfw8d	2/2	Running	0	5d5h

Figure 5.4: All pods, including Istio service mesh

Now, we can see that *Istio-system* namespace have appeared with a bunch of service mesh system pods and are efficiently running. Secondly, we will inject the Istio sidecars into the Bookinfo application pods. For this purpose we will use the `inject` command of our `auto-istio.sh` script. After the `inject` command we should specify the required namespace, in our case the *default* one. Accordingly, the command looks like this: `./auto-istio.sh inject default`. Figure 5.5 represents the state of the *default* namespace after injection.



```
root@dev-KVM:~# kubectl get pods -l app=productpage
NAME                              READY   STATUS    RESTARTS   AGE
productpage-v1-7f44c4d57c-bc26j   2/2     Running   0          35s
productpage-v1-7f44c4d57c-bq8hw   2/2     Running   0          35s
productpage-v1-7f44c4d57c-db2zh   2/2     Running   0          35s
productpage-v1-7f44c4d57c-hghhh   2/2     Running   0          43h
productpage-v1-7f44c4d57c-lv4bq   2/2     Running   0          35s
productpage-v1-7f44c4d57c-mggx5   2/2     Running   0          35s
productpage-v1-7f44c4d57c-mpghl   2/2     Running   0          8m23s
productpage-v1-7f44c4d57c-p44n2   2/2     Running   0          35s
productpage-v1-7f44c4d57c-qb92b   2/2     Running   0          35s
productpage-v1-7f44c4d57c-x8q9w   2/2     Running   0          8m23s
```

Figure 5.5: The application pods after Istio injection

Now, we can see that every application's pod is running two containers.

Istio just embedded the *Istio-proxy* sidecar inside of every pod of the *default* namespace.

Finally, we will finish the solution deployment by setting the peer authentication rule for our application. To accomplish this, we will use the `secure` command of our `auto-istio.sh` script. As in the case of injection, we should also specify the namespace for the rule creation. Accordingly, the command looks like this: `./auto-istio.sh secure default`. To check the status of the created rule we will use the following command `kubectl -n default describe peerauthentication`. Figure 5.6 illustrates the description of the just created authentication rule.

```
Namespace:    default
Labels:       <none>
Annotations:  API Version:  security.istio.io/v1beta1
Kind:         PeerAuthentication
Metadata:
  Creation Timestamp:  2020-05-29T17:53:12Z
  Generation:          1
  Managed Fields:
    API Version:  security.istio.io/v1beta1
    Fields Type:  FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          .:
          f:kubectl.kubernetes.io/last-applied-configuration:
      f:spec:
        .:
        f:mtls:
          .:
          f:mode:
    Manager:         kubectl
    Operation:       Update
    Time:            2020-05-29T17:53:12Z
  Resource Version:  1134260
  Self Link:         /apis/security.istio.io/v1beta1/namespaces/default/peerauthentications/default
  UID:               63475050-e6ea-47b8-b419-f3c5544583c8
Spec:
  Mtls:
    Mode:  STRICT
Events:    <none>
```

Figure 5.6: The peer authentication rule description

We can observe that the authentication rule is successfully created and set in STRICT mode, which compels the pod to receive only an mTLS traffic.

Therefore, we have effortlessly deployed and configured the Istio service mesh for secure communication between microservices of the Bookinfo application. Compared with the initial architecture of the application, represented by Figure 3.1, the current architecture after of our solution implementation is presented in Figure 5.7.

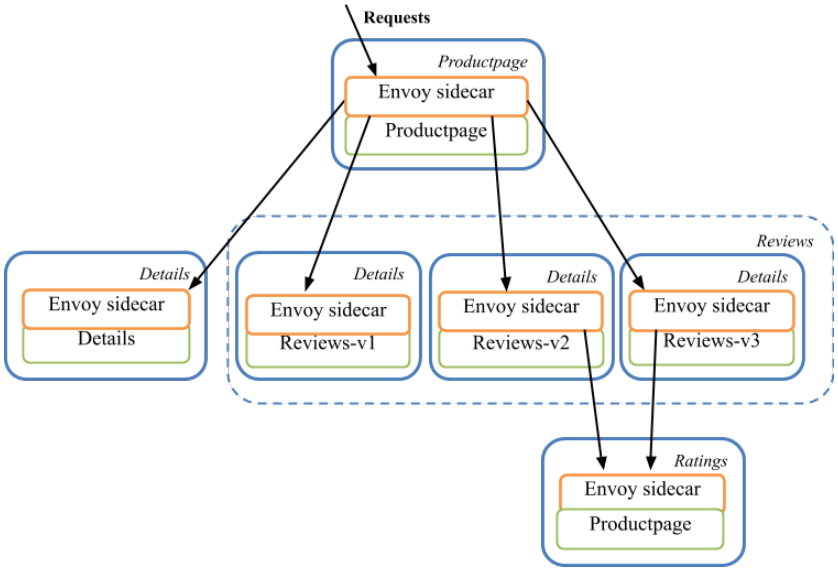


Figure 5.7: The peer authentication rule description

As we can see, the structure of the application components and service communication became different. The communication between services now is handled through Istio sidecars, while all external incoming requests are passing through the Istio ingress gateway. Thus, Istio service mesh controls internal and external traffic via Envoy sidecars and Ingress/Egress policies.

Chapter 6

Evaluation

In the previous section 5, we showed the deployment process of our solution and configured it for secure mTLS communication. This section will analyze the deployed solution for conformity with requirements developed in section 1.3. We will analyze them in more detail in the following subsections. The chapter will be concluded with the revised relevance of threats, taking into consideration the deployed and analyzed solution.

6.1 Requirement analysis

6.1.1 Secure communication

As we mentioned in previous sections, Istio service mesh implements the mutual TLS protocol to secure communications between microservices. This means that throughout the whole Kubernetes network, secured with mTLS pods will not be able to communicate with any other pods. The following example will help to prove this. It contains useful and efficient diagnostic tools for TLS and SSL analysis. To confirm that our communications indeed are realized through mTLS, we will use the openssl to connect to *productpage:9080* address from the istio-proxy container of any application service. To perform this, we will use the following command:

```
kubectl exec -it $(kubectl get pod -l app=ratings -o jsonpath
='.items[0].metadata.name')" -c istio-proxy -- openssl s_client
-connect productpage:9080 > productpage-ope nssl.txt
```

As the result, we will receive the openssl report about the TLS connection, including certificates, signature algorithms, TLS version, key characteristics,

etc. The openssl report shows that communication is indeed secure with TLSv1.3, uses 2048 bit public key, and the chain of certificates. The full output of the request is represented in Appendix B.

Additionally, we will check the availability of service with enabled and disabled mTLS mode. For example, we will send to the **Productpage** service a simple API request from the **Ratings** service via the following command:

```
root@dev-KVM:/home/eearklu# kubectl exec $(kubectl get
pod -l app=ratings -n default -o jsonpath={.items..meta
data.name}) -c ratings -n default -- curl productpage:9
080/productpage -s -o /dev/null -w "%{http_code}\n"
200
```

Figure 6.1: The command for API request with OK respond

As we can see on the Figure 6.1, the service responded with the code **200**, which means that the request was successful. Now, we will create a peer authentication rule for **Productpage** service that will disable the mTLS support on it, with the following code:

```
cat <<EOF | kubectl apply -n default -f -
    apiVersion: "security.istio.io/v1beta1"
    kind: "PeerAuthentication"
    metadata:
        name: "productpage"
        namespace: "default"
    spec:
        selector:
            matchLabels:
                app: productpage
        mtls:
            mode: DISABLE
EOF
```

Now, when the peer authentication rule disabled the mTLS support, the **Productpage** service should be unable to communicate with the rest of the Bookinfo services. To confirm this, we again will send the API request from the **Ratings** to the **Productpage** service.

Figure 6.2 shows that the request failed with the Service Unavailable 503 code. Thus, we just proved that our services are protected and will not accept any requests without the mTLS protocol.

```
root@dev-KVM:~# kubectl exec $(kubectl get pod -l app=r
atings -n default -o jsonpath={.items..metadata.name})
-c ratings -n default -- curl productpage:9080/productp
age -s -o /dev/null -w "%{http_code}\n"
503
```

Figure 6.2: The command for API request with Service Unavailable respond

In order to properly provide the communication by mTLS, Istio uses transparent encryption and certificate management system. Following subsections will ensure that encryption and certificate distribution are implemented and efficiently works.

6.1.2 Encryption

To ensure that the traffic is indeed encrypted we will the request from one service to another and verify the traffic via `tcpdump` program. To perform it we will access the **productpage** container and execute there the `tcpdump` to listen the 9080 port for traffic. Afterward, in the other terminal, we will send the API request to the **Productpage** service. When the request is sent, we can observe the received by **productpage** traffic in the previous terminal. The result of this implementation process is illustrated in Figure 6.3.

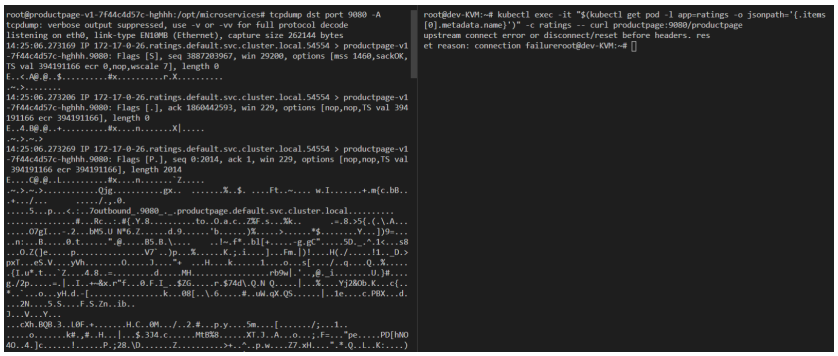


Figure 6.3: The result of receiving of the encrypted traffic

The Figure points to the conclusion that **Productpage** indeed received the encrypted traffic. Thus, we can safely state that the Istio service mesh provides efficient encryption of the mTLS traffic.

6.1.3 Certificate life-cycle management

To ensure that the Istio certificate management system efficiently create and distribute robust certificates, we will try to reach them through the Istio proxy containers via `openssl` software.

For example, we can access the **Productpage** service via API request and observe if the response will return the certificate chain. In this case, we can send the API request to the `productpage:9080` from any other service pod. Since we have already sent such a request in the Section 5.2, we can find the encrypted certificate chain in Appendix B as well. To decrypt certificates, we will use the commands illustrated in Figure 6.4.

```
root@dev-KVM:/tmp/certs# sed -n '/-----BEGIN CERTIFICATE-----/{:start /----
-END CERTIFICATE-----/!{N;b start};/.*/p}' productpage-proxy-cert.txt > cer
ts.pem
root@dev-KVM:/tmp/certs# awk 'BEGIN {counter=0;} /BEGIN CERT/{counter++} {
print > "proxy-cert-" counter ".pem"}' < certs.pem
root@dev-KVM:/tmp/certs#
root@dev-KVM:/tmp/certs# openssl x509 -in ./proxy-cert-1.pem -text -noout >
 ./pod-cert-1.crt.txt
root@dev-KVM:/tmp/certs# openssl x509 -in ./proxy-cert-2.pem -text -noout >
 ./pod-cert-2.crt.txt
```

Figure 6.4: The certificate decryption commands

The `sed` command is a Linux stream editor. In our case, `sed` allows us to parse the certificates into a separate `certs.pem` file by a provided template. Then we use the `awk` utility to separate certificates into different files. Finally, we read each certificate with openssl and save the decrypted certificates into `proxy-cert-1.pem` and `proxy-cert-2.pem` respectively. The decrypted certificates are presented in Appendix C. As the result, we observe that decrypted certificates are robust and properly managed.

6.1.4 Automation

To automate our solution we created the script, the deployment process of which was represented in the previous section. To measure the execution speed of it, we will repeat the deployment process via the Linux `time` utility. The results are represented in Figure 6.5.

The sum of the script execution time results is **82.025** seconds. During this time the script installed Istio service mesh into the system injected the application pods with Istio sidecars and secured the communication between them by creating a peer authentication rule. It is a considerable result, which shows that a developer without a deep understanding of a service

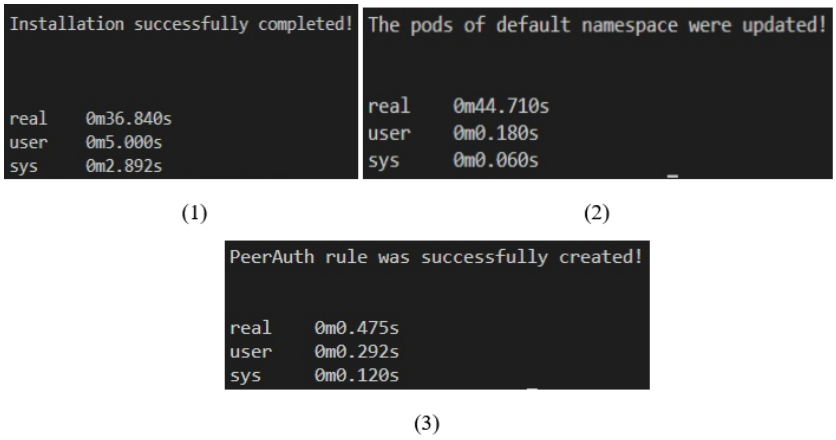


Figure 6.5: The execution results of the following script commands: (1) - `install`, (2) - `inject`, (3) - `secure`

mesh technology can rapidly deploy the solution and secure the microservices communication of the Kubernetes environment.

6.1.5 Scalability

The Istio service mesh allows to label a namespace in order to attach an Istio sidecar to every pod inside of such namespace. During the Deployment section implementation, we already labeled the **default** namespace with the Istio label for automatic injection via the following command:

```
kubectl label namespace default istio-injection=enabled
```

Thus, the default namespace is already configured for a scalable sidecar injection. To confirm the solution's scalability, we will create replicas of our application services. To generate the 9 more `Productpage` pod replicas, we will execute the following Kubernetes command:

```
kubectl scale deployments/productpage-v1 --replicas=10
```

Figure 6.6 presents the result of the **Productpage** pod scaling.

The figure point to a conclusion that all appeared replicas of the **Productpage** pod have two containers inside, which are the application container and the Istio sidecar.

Thus, Istio provides high scalability by dynamically injecting sidecars

```
root@dev-KVM:~# kubectl get pods -l app=productpage
NAME                              READY   STATUS    RESTARTS   AGE
productpage-v1-7f44c4d57c-bc26j   2/2     Running   0          35s
productpage-v1-7f44c4d57c-bq8hw   2/2     Running   0          35s
productpage-v1-7f44c4d57c-db2zh   2/2     Running   0          35s
productpage-v1-7f44c4d57c-hghhh   2/2     Running   0          43h
productpage-v1-7f44c4d57c-lv4bq   2/2     Running   0          35s
productpage-v1-7f44c4d57c-mggx5   2/2     Running   0          35s
productpage-v1-7f44c4d57c-mpghl   2/2     Running   0          8m23s
productpage-v1-7f44c4d57c-p44n2   2/2     Running   0          35s
productpage-v1-7f44c4d57c-qb92b   2/2     Running   0          35s
productpage-v1-7f44c4d57c-x8q9w   2/2     Running   0          8m23s
```

Figure 6.6: The set of the **Productpage** replicas

into any number of pods. The injected sidecars are deployed as a new container, thereby providing secure communication without any changes to the application code.

6.1.6 Performance

To analyze the system performance parameters, we used the `Prometheus` software for the data collection and `Grafana` software for the data representation. The state of the system before the solution deployment is represented with Grafana and shown on the Figure 6.7.

Figure 6.7: The system performance indicators before the solution implementation

The resource usage characteristics are adequate and expected accordingly to our machine attributes. It is not using too much CPU and memory for idle work and the filesystem usage is also proper due to the machine had 20 Gb of free space before the working environment installation.

The Figure 6.7 displays the system performance after the solution deployment.

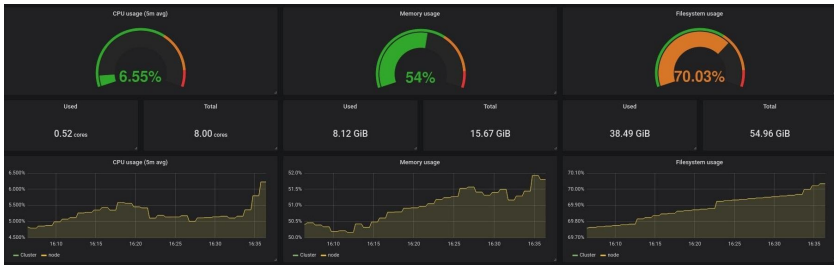


Figure 6.8: The system performance indicators after the solution implementation

Compared with the initial state, the system usage is slightly increased. The CPU usage rose on **1.63%**, the memory usage on **2%**, and filesystem usage increased on **2.85%**. This insignificant increment will not affect dramatically on the system, especially on a scale of large company equipment.

6.2 Threat solving

As we mentioned in Section 4.2, the devised Kubernetes threat model consists of 6 security threats in the scope of internal communication of microservices. Based on the above solution analysis results, we can infer what threats are addressed by our designed method. Table 6.1 represents the threats and countermeasures provided by our solution.

The threat of a Man-in-the-Middle attack implies the interception of an authentication request by an external attacker in order to impersonate the recipient and get the sensitive data that was intended for the true recipient. The implementation of a certificate management system allows for distributing reliable certificates for every service. Therefore, when a service wants to share data with another service, it can confirm the recipient's identity by its certificate as the recipient can confirm the identity of the sender by its certificate. Such certificate sharing is part of the mutual TLS handshake process, which occurs when services share information for the first time.

During the sniffing attack, the transmitting data can be intercepted and disclosed by an external attacker. In terms of Kubernetes, sniffing can be executed both on a network and application level. On the network level, data could be sniffed during the data transmission between nodes or clusters. On the application level, the internal traffic between services can be intercepted from a compromised container. In order to save data confidentiality, Istio encrypts all transmitting data between sidecars by transparent

	Threat	Solution	Effect
1	Man-in-the-Middle attack between cluster components	Certificate management system	Prevented
2	Sniffing of internal traffic	Transparent TLS encryption	Prevented
3	Unauthorized connection to another pods and applications API	Service mesh Envoy sidecar with egress policies	Mitigated
4	Exploiting the access to Kubelet API	Secure communication through mutual TLS	Mitigated
5	Network scanning	Service mesh Envoy sidecar with egress policies	Mitigated
6	Changing the VM settings through compromised container with root privileges	No	Not prevented

Table 6.1: Threat model

TLS encryption.

If an attacker compromised a pod, he can try to connect and compromise other pods. As with network scanning situation, if the communication between services is not restricted, the attacker could access any service through the compromised one. However, the Istio service mesh architecture allows to set restrictions between services via policies, thereby significantly complicate the compromising of other services and limiting the attack area of the compromised pod. Such a solution also helps to restrict the requesting of applications API and API of some system components, such as kubelet.

The threat network scanning does not harm to confidentiality, availability, or integrity of a system and is even not considered as a threat. However, the network scan can give an attacker a great advantage and help to prepare for a real attack. An attacker can scan a network for services and pods names, IP addresses, and ports and devise its next attack more accurately. The Envoy proxy sidecars can restrict outgoing service requests via extended by Istio egress policies, thereby preventing the request dispatch to restricted services.

However, not all threats could be prevented or mitigated by the service mesh solution. If an attacker could compromise a container with root privileges he could try to reach the host machine and change the VM settings. Thus, he could disrupt the work of this VM and all pods that are running there. This threat is poorly explored and difficult to realize, however it can significantly harm the Kubernetes structure.

Chapter 7

Conclusions

Currently, scientific progress is becoming faster year by year. In recent years, the monolithic architecture of applications has been gradually replacing with microservice architecture. Microservices decompose an application into a set of manageable services that are much faster to develop, and much easier to understand and maintain. In 2015, Google presents the Kubernetes open-source platform, which allows to manage containers and services and define configuration and automation. However, parallelly with the progress of IT technologies, the security issue is also progressing. New technologies require new security mechanisms, as with Kubernetes release. Despite the extensive and complex Kubernetes security mechanisms, some security issues, as the lack of internal communication security, are still relevant.

The goal of the thesis, which is to decrease the number of threats, do not significantly degrade the system's performance, and substantially simplify the deployment process, was achieved.

Were reached the following thesis tasks:

a. Review and analysis of literary sources and existing solutions in the field of existing options for ensuring the safety of microservice communication
b. Development of requirements for implementation in a corporate system of a method for ensuring the safety of microservice communications.
c. Development of a method for securing microservice communications within a Kubernetes virtual environment.
d. Conduct experimental studies to evaluate the effectiveness of the developed method

Bibliography

[1] Aniszczyk, C. Open sourcing the kubernetes security audit. https://www.cncf.io/blog/2019/08/06/open-sourcing-the-kubernetes-security-audit/. Accessed 01.06.2020.

[2] Authors, I. Bookinfo application. https://istio.io/docs/examples/bookinfo/. Accessed 01.06.2020.

[3] Authors, I. The official documentation of istio software. https://istio.io/docs/concepts/. Accessed 01.06.2020.

[4] Authors, L. The official documentation of linkerd software. https://linkerd.io/2/overview/. Accessed 01.06.2020.

[5] Authors, T. K. The official documentation of kubernetes software. https://kubernetes.io/docs/concepts/. Accessed 01.06.2020.

[6] Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[7] Boyle, J. M., Maiwald, E. S., and Snow, D. W. Apparatus and method for providing network security, Aug. 17 1999. US Patent 5,940,591.

[8] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *Queue 14*, 1 (2016), 70–93.

[9] Carretero, J., and Blas, J. G. Introduction to cloud computing: platforms and solutions. *Cluster computing 17*, 4 (2014), 1225–1229.

[10] Chandramouli, R. Security strategies for microservices-based application systems. Tech. rep., 2019.

[11] Corporation, T. M. Open sourcing the kubernetes security audit. `https://attack.mitre.org/matrices/enterprise/`. Accessed 01.06.2020.

[12] Das, M. L., and Samdaria, N. On the security of ssl/tls-enabled applications. *Applied Computing and informatics 10*, 1-2 (2014), 68–81.

[13] Dewi, L. P., Noertjahyana, A., Palit, H. N., and Yedutun, K. Server scalability using kubernetes. In *2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON)* (2019), IEEE, pp. 1–4.

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

[15] Dillon, T., Wu, C., and Chang, E. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications* (2010), Ieee, pp. 27–33.

[16] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[17] Edwards, S. Threat model. `https://github.com/kubernetes/community/blob/master/wg-security-audit/findings/Kubernetes%20Threat%20Model.pdf`. Accessed 01.06.2020.

[18] Esposito, C., Castiglione, A., and Choo, K.-K. R. Challenges in delivering software in the cloud as microservices. *IEEE Cloud Computing 3*, 5 (2016), 10–14.

[19] Grobauer, B., Walloschek, T., and Stocker, E. Understanding cloud computing vulnerabilities. *IEEE Security & privacy 9*, 2 (2010), 50–57.

[20] Hightower, K., Burns, B., and Beda, J. *Kubernetes: up and running: dive into the future of infrastructure.* " O'Reilly Media, Inc.", 2017.

[21] Hussain, F., Li, W., Noye, B., Sharieh, S., and Ferworn, A. Intelligent service mesh framework for api security and management. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)* (2019), IEEE, pp. 0735–0742.

[22] Inukollu, V. N., Arsi, S., and Ravuri, S. R. Security issues associated with big data in cloud computing. *International Journal of Network Security & Its Applications 6*, 3 (2014), 45.

[23] Kang, M., Shin, J.-S., and Kim, J. Protected coordination of service mesh for container-based 3-tier service traffic. In *2019 International Conference on Information Networking (ICOIN)* (2019), IEEE, pp. 427–429.

[24] Khatri, A., Khatri, V., Nirmal, D., Pirahesh, H., and Herness, E. *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Packt Publishing, 2020. `https://books.google.fi/books?id=Mg3aDwAAQBAJ/`. Accessed 01.06.2020.

[25] Li, W., Lemieux, Y., Gao, J., Zhao, Z., and Han, Y. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)* (2019), IEEE, pp. 122–1225.

[26] Manu, A., Patel, J. K., Akhtar, S., Agrawal, V., and Murthy, K. B. S. Docker container security via heuristics-based multilateral security-conceptual and pragmatic study. In *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)* (2016), IEEE, pp. 1–14.

[27] Medel, V., Rana, O., Bañares, J. Á., and Arronategui, U. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing* (2016), pp. 257–262.

[28] Mell, P., Grance, T., et al. The nist definition of cloud computing.

[29] Modak, A., Chaudhary, S., Paygude, P., and Ldate, S. Techniques to secure data on cloud: Docker swarm or kubernetes? In *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)* (2018), IEEE, pp. 7–12.

[30] Namiot, D., and Sneps-Sneppe, M. On micro-services architecture. *International Journal of Open Information Technologies 2*, 9 (2014), 24–27.

[31] Prince, J. D. Introduction to cloud computing. *Journal of Electronic Resources in Medical Libraries 8*, 4 (2011), 449–458.

[32] Sayfan, G. *Mastering kubernetes*. Packt Publishing Ltd, 2017.

[33] Sheikh, O., Dikaleh, S., Mistry, D., Pape, D., and Felix, C. Modernize digital applications with microservices management using the istio service mesh. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering* (2018), pp. 359–360.

[34] Sun, Y., Nanda, S., and Jaeger, T. Security-as-a-service for microservices-based cloud applications. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)* (2015), IEEE, pp. 50–57.

[35] Torkura, K. A., Sukmana, M. I., and Meinel, C. Integrating continuous security assessments in microservices and cloud native applications. In *Proceedings of the10th International Conference on Utility and Cloud Computing* (2017), pp. 171–180.

[36] Vayghan, L. A., Saied, M. A., Toeroe, M., and Khendek, F. Deploying microservice based applications with kubernetes: experiments and lessons learned. In *2018 IEEE 11th international conference on cloud computing (CLOUD)* (2018), IEEE, pp. 970–973.

[37] Weizman, Y. Threat matrix for kubernetes. `https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/`. Accessed 01.06.2020.

[38] Wu, H., Ding, Y., Winer, C., and Yao, L. Network security for virtual machine in cloud computing. In *5th International Conference on Computer Sciences and Convergence Information Technology* (2010), IEEE, pp. 18–21.

[39] Yarygina, T., and Bagge, A. H. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)* (2018), IEEE, pp. 11–20.

[40] Zissis, D., and Lekkas, D. Addressing cloud computing security issues. *Future Generation computer systems 28*, 3 (2012), 583–592.

Appendix A

The automation script

```
#!/bin/bash
# Istio Deployment script

# Setting variables
operation="$1"
namespace="$2"
deployment="$3"

case $operation in
    install)
        # Downloading and extracting the latest release
        echo -e "\nDownloading and extracting the latest release...\n"
        curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.5.2 sh -
        if [ "$?" = "0" ]; then
            echo -e "\nFiles downloaded successfully!\n"
        else
            echo -e "\nDownlading error. Interrupting deploying\n"
            exit 1
        fi

        # Adding the istioctl client to the environmental variable PATH
        echo -e "\nAdding the istioctl client to the environmental variable PATH...\n"
        export PATH=$PWD/istio-1.5.2/bin:$PATH
        if [ "$?" = "0" ]; then
            echo -e "\nPATH variable updated!\n"
        else
            echo -e "\nDeclaration error. Interrupting deploying\n"
            exit 1
        fi

        # Installing Istio with the demo configuration profile
        echo -e "\nInstalling Istio with the demo configuration profile...\n"
        istioctl manifest apply --set profile=demo
        if [ "$?" = "0" ]; then
            echo -e "\nIstio was deployed successfully!\n"
        else
            echo -e "\nDeployment error. Interrupting deploying\n"
            exit 1
        fi

        echo -e "\n\nInstallation successfully completed!\n\n"
    ;;
    inject)
        if [ -z $namespace ]; then
            echo "Please specify the namespace!"
        else
            if [ -z $deployment ]; then
                echo -e "Deployment was not provided\n"

                # Injecting sidecar into the user's namespace
                echo -e "\nAutomatizing sidecar injection in the $namespace namespace
                    ...\n"
                kubectl label namespace $name istio-injection=enabled

                if [ "$?" = "0" ]; then
                    echo -e "\nLabel was added successfully!"
                else
```

```
                    echo -e "\nDeployment error. Interrupting deploying\n"
                    exit 1
                fi
            else
                # Injecting sidecar into the user's deployment
                echo -e "\nAutomatizing sidecar injection in the $deployment deployment
                     of the $namespace namespace...\n"
                kubectl get deployment $namespace -o yaml | istioctl kube-inject -f - |
                     kubectl apply -f -
            fi
        fi
    ;;
    secure)
        echo -e "\nVerifying the peer authentication policies in the system...\n"
        kubectl get peerauthentication --all-namespaces

        echo -e "\nVerifying the destination rules for services in all namespaces...\n"
        kubectl get destinationrules.networking.istio.io --all-namespaces -o yaml | grep
             "host:"
        if [ "$?" = "0" ]; then
            echo -e "\nFound existing authentication policies."
            echo -e "Please, be careful."
        else
            echo -e "\nNo destination rules was found.\n"
        fi

        if [ -z $namespace ]; then
            echo -e "Please specify the namespace or 'system' entity!\n"
        else
            if [ "$namespace" = "system" ]; then
                echo -e "\nSetting Peer Authentication Rule for the whole cluster...\n"
                kubectl apply -n istio-system -f - <<EOF
                apiVersion: "security.istio.io/v1beta1"
                kind: "PeerAuthentication"
                metadata:
                    name: "system"
                spec:
                    mtls:
                        mode: STRICT
EOF
                if [ "$?" = "0" ]; then
                    echo -e "\nPeerAuth rule was successfully created!\n"
                else
                    echo -e "\nPeerAuth rule creation error\n"
                fi
            else
                echo -e "\nSetting Peer Authentication Rule for the $namespace namespace
                     ...\n"
                kubectl apply -n $namespace -f - <<EOF
                apiVersion: "security.istio.io/v1beta1"
                kind: "PeerAuthentication"
                metadata:
                    name: "$namespace"
                spec:
                    mtls:
                        mode: STRICT
EOF
                if [ "$?" = "0" ]; then
                    echo -e "\nPeerAuth rule was successfully created!\n"
                else
                    echo -e "\nPeerAuth rule creation error\n"
                fi
            fi
        fi
    ;;
    *)
        echo -e "\nPlease, enter any operation (install, inject, secure)\n"
    ;;
esac
```

Appendix B

The TCPDump response

```
CONNECTED(00000005)
depth=1 O = cluster.local
verify error:num=19:self signed certificate in certificate chain
---
Certificate chain
 0 s:
   i:O = cluster.local
-----BEGIN CERTIFICATE-----
MIIDLDCCAhSgAwIBAgIQWssp4xizwokpfLxEzCE4KzANBgkqhkiG9w0BAQsFADAY
MRYwFAYDVQQKEwljbHVzdGVyLmxvY2FsMB4XDTIwMDUyODA3MzEwN1oXDTIwMDUy
OTA3MzEwN1owADCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALMMfgOM
AmzxTkRAGmZNFoiLWNzSdkzb0zjH7RrOZDWt7AbYrNO/0U3zNiUrjE0nAFeIrIoA
6Ry9aqNY+EvAiAE+ZXqwHOKfM39i2IPW1Mq7YRlp6E0vAsjS0NKh9oilxX9xbBPa
htZ9MUjj01orR9ABRHGRoxV7R1cbQrYIhMmEZcjhBua5McmSrcWuQFQDvyi6/gE3
C3ivEClLqzyjkdPsvtfVOqtMKVeOgJ2XWoSoxOhunNnZB63bmdKwvls+Of/p+Bwc
N0JCq4Cq3mdu/sQNcbP4vkWBgj+ZbvCkZyQ0lQj6YO8dTIiWfoYiIU4wZ+fH7fc7
F04jT9DN8o1sX7MCAwEAAaOBiTCBhjAOBgNVHQ8BAf8EBAMCBaAwHQYDVR0lBBYw
FAYIKwYBBQUHAwEGCCsGAQUFBwMCMAwGA1UdEwEB/wQCMAAwRwYDVR0RAQH/BD0w
O4Y5c3BpZmZlOi8vY2x1c3Rlci5sb2NhbC9ucy9kZWZhdWx0L3NhL2Jvb2tpbmZv
LXByb2R1Y3RwYWdlMA0GCSqGSIb3DQEBCwUAA4IBAQAthXUvhieZYUYraafyl2gW
EYA2aohZ/BjpkYTluwPXMMgLPcRUP/Oz+k+b+iTKgt24GKryizQENKUkZJdkNtbA
6Nms8NACnwXda2aN/KCRrb/fbFabfaVo1GVM0MQog5JAfS97Z7VULNVGfFv6pY5Y
L2+M4oR6QmHxguIYXinXrF8+HWG12Mg5bwnuw7sn74B7tDKhptLh2ajlOlMYj/xN
uejXYplw7uWPRZKNPTZq1i+JKzRy4BP0AjDhtdS6P3PiFGzZor8sAe7SWE0xpcsu
alFM2CuSWrF4vaZl/VO5DBlU+pBFhvmFD+pQ6LryxCg8MtpfQZyqlhUcOnat01p2
-----END CERTIFICATE-----
 1 s:O = cluster.local
   i:O = cluster.local
-----BEGIN CERTIFICATE-----
MIIC3jCCAcagAwIBAgIRAP+uRdSyUv0ih6XJHDjJuZwwDQYJKoZIhvcNAQELBQAw
GDEWMBQGA1UEChMNY2x1c3Rlci5sb2NhbDAeFw0yMDA1MjcxODU0NTVaFw0zMDA1
MjUxODU0NTVaMBgxFjAUBgNVBAoTDWNsdXN0ZXIubG9jYWwwggEiMA0GCSqGSIb3
DQEBAQUAA4IBDwAwggEKAoIBAQDKbaIkMRtp/v//QfjQ7nWwQ1no4lDQB82cwq0j
wId/ysJYY8CVNjtiAsJRPipuUzVxwNlR3nufDMjzL5KsvBA1nIpSUODhB/TAxRok
F3ravVCs0pIEOEXBQcD+i0HIJAhCMpqWQ4j3KfbikyDQDn7qyVepjplMmCddSEgX
H6++kBPDkd15/aGqf7qN/RtSs73CulCwGdQsl1U+bPsEXH0CvIlaAqeTntwAmUWF
pHxL9UDL+dpEVJZvPCo5NTtpSkDSn6+R4SQpkZsSDN4GXgShzFGCunZImf7Dy1PD
AzwqEMXDUs5N+XnA06lkxkWYzGAd9KWmr3K/0OX4/nqqavtrAgMBAAGjIzAhMA4G
A1UdDwEB/wQEAwICBDAPBgNVHRMBAf8EBTADAQH/MA0GCSqGSIb3DQEBCwUAA4IB
AQDCaeBLL+cY2ZtB/ZhnHHz3IM/sMGzrlcLEZVNuAYbzFIEKE90JtZ5pYpjt+L5C
R6tOdYfcZaXYxFXqYqfte+iOdLev0cRAXxEbV1z0XfwZ5laVXbPr2Cfv0OVPGu5r
tMGYWG2RfLVIfpcdU/eSWiab1D0nzd/Dx59FLfMoSqKrzqiyUKmVaKm9V2+0LZzm
1cPkqJmkbb6vAyGRj+qnFOeLnkVyc1d4+B4n9xcTdSNpjExcG8fm2QlRnda8Vr5w
x44SYeGDZiTQWaFiPOclnI0Z+LZspZK8lT+2YgqjPyUNvcJlaUtMWHHZYVBrqZ6Q
ttpAWPE00NjSALHqJ7KXfT8R
-----END CERTIFICATE-----
---
Server certificate
subject=

issuer=O = cluster.local

---
Acceptable client certificate CA names
O = cluster.local
Requested Signature Algorithms: ECDSA+SHA256:RSA-PSS+SHA256:RSA+SHA256:ECDSA+SHA384:RSA-
      PSS+SHA384:RSA+SHA384:RSA-PSS+SHA512:RSA+SHA512:RSA+SHA1
```

```
Shared Requested Signature Algorithms: ECDSA+SHA256:RSA-PSS+SHA256:RSA+SHA256:ECDSA+
    SHA384:RSA-PSS+SHA384:RSA+SHA384:RSA-PSS+SHA512:RSA+SHA512
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 2114 bytes and written 423 bytes
Verification error: self signed certificate in certificate chain
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 19 (self signed certificate in certificate chain)
---
139963402621376:error:1409445C:SSL routines:ssl3_read_bytes:tlsv13 alert certificate
    required:../ssl/record/rec_layer_s3.c:1528: SSL alert number 116
```

Appendix C

The decrypted certificates

```
Certificate 1:
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            5a:cb:29:e3:18:b3:c2:89:29:7c:bc:44:cc:21:38:2b
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: O=cluster.local
        Validity
            Not Before: May 28 07:31:07 2020 GMT
            Not After : May 29 07:31:07 2020 GMT
        Subject:
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:b3:0c:7e:03:8c:02:6c:f1:4e:44:40:1a:66:4d:
                    16:88:8b:58:dc:d2:76:4c:db:d3:38:c7:ed:1a:ce:
                    64:35:ad:ec:06:d8:ac:d3:bf:d1:4d:f3:36:25:2b:
                    8c:4d:27:00:57:88:ac:8a:00:e9:1c:bd:6a:a3:58:
                    f8:4b:c0:88:01:3e:65:7a:b0:1c:e2:9f:33:7f:62:
                    d8:83:d6:d4:ca:bb:61:19:69:e8:4d:2f:02:c8:d2:
                    d0:d2:a1:f6:88:a5:c5:7f:71:6c:13:da:86:d6:7d:
                    31:48:e3:d3:5a:2b:47:d0:01:44:71:91:a3:15:7b:
                    47:57:1b:42:b6:08:84:c9:84:65:c8:e1:06:e6:b9:
                    31:c9:92:ad:c5:ae:40:54:03:bf:28:ba:fe:01:37:
                    0b:78:af:10:29:4b:ab:3c:a3:91:d3:ec:be:d7:d5:
                    3a:ab:4c:29:57:8e:80:9d:97:5a:84:a8:c4:e8:6e:
                    9c:d9:d9:07:ad:db:99:d2:b0:be:5b:3e:39:ff:e9:
                    f8:1c:1c:37:42:42:ab:80:aa:de:67:6e:fe:c4:0d:
                    71:b3:f8:be:45:81:82:3f:99:6e:f0:a4:67:24:34:
                    95:08:fa:60:ef:1d:4c:88:96:7e:86:22:21:4e:30:
                    67:e7:c7:ed:f7:3b:17:4e:23:4f:d0:cd:f2:8d:6c:
                    5f:b3
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Alternative Name: critical
                URI:spiffe://cluster.local/ns/default/sa/bookinfo-productpage
    Signature Algorithm: sha256WithRSAEncryption
         2d:85:75:2f:86:27:99:61:46:2b:69:a7:f2:97:68:16:11:80:
         36:6a:88:59:fc:18:e9:91:84:e5:bb:03:d7:30:c8:0b:3d:c4:
         54:3f:f3:b3:fa:4f:9b:fa:24:ca:82:dd:b8:18:aa:f2:8b:34:
         04:34:a5:24:64:97:64:36:d6:c0:e8:d9:ac:f0:d0:02:9f:05:
         dd:6b:66:8d:fc:a0:91:ad:bf:df:6c:56:9b:7d:a5:68:d4:65:
         4c:d0:c4:28:83:92:40:7d:2f:7b:67:b5:54:2c:d5:46:7c:5b:
         fa:a5:8e:58:2f:6f:8c:e2:84:7a:42:61:f1:82:e2:18:5e:29:
         d7:ac:5f:3e:1d:61:b5:d8:c8:39:6f:09:ee:c3:bb:27:ef:80:
         7b:b4:32:a1:a6:d2:e1:d9:a8:e5:3a:53:18:8f:fc:4d:b9:e8:
         d7:62:99:70:ee:e5:8f:45:92:8d:3d:36:6a:d6:2f:89:2b:34:
         72:e0:13:f4:02:30:e1:b5:d4:ba:3f:73:e2:14:6c:d9:a2:bf:
         2c:01:ee:d2:58:4d:31:a5:cb:2e:6a:51:4c:d8:2b:92:5a:b1:
```

```
        78:bd:a6:65:fd:53:b9:0c:19:54:fa:90:45:86:f9:85:0f:ea:
        50:e8:ba:f2:c4:28:3c:32:da:5f:41:9c:aa:96:15:1c:3a:76:
        ad:d3:5a:76


Certificate 2:
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            ff:ae:45:d4:b2:52:fd:22:87:a5:c9:1c:38:c9:b9:9c
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: O=cluster.local
        Validity
            Not Before: May 27 18:54:55 2020 GMT
            Not After : May 25 18:54:55 2030 GMT
        Subject: O=cluster.local
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:ca:6d:a2:24:31:1b:69:fe:ff:ff:41:f8:d0:ee:
                    75:b0:43:59:e8:e2:50:d0:07:cd:9c:c2:ad:23:c0:
                    87:7f:ca:c2:58:63:c0:95:36:3b:62:02:c2:51:3e:
                    2a:6e:53:35:71:c0:d9:51:de:7b:9f:0c:c8:f3:2f:
                    92:ac:bc:10:35:9c:8a:52:50:e0:e1:07:f4:c0:c5:
                    1a:24:17:7a:da:bd:50:ac:d2:92:04:38:45:c1:41:
                    c0:fe:8b:41:c8:24:08:42:32:9a:96:43:88:f7:29:
                    f6:e2:93:20:d0:0e:7e:ea:c9:57:a9:8e:99:4c:98:
                    27:5d:48:48:17:1f:af:be:90:13:c3:91:dd:79:fd:
                    a1:aa:7f:ba:8d:fd:1b:52:b3:bd:c2:ba:50:b0:19:
                    d4:2c:97:55:3e:6c:fb:04:5c:7d:02:bc:89:5a:02:
                    a7:93:9e:dc:00:99:45:85:a4:7c:4b:f5:40:cb:f9:
                    da:44:54:96:6f:3c:2a:39:35:3b:69:4a:40:d2:9f:
                    af:91:e1:24:29:91:9b:12:0c:de:06:5e:04:a1:cc:
                    51:82:ba:76:48:99:fe:c3:cb:53:c3:03:3c:2a:10:
                    c5:c3:52:ce:4d:f9:79:c0:d3:a9:64:c6:45:98:cc:
                    60:1d:f4:a5:a6:af:72:bf:d0:e5:f8:fe:7a:aa:6a:
                    fb:6b
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Certificate Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
         c2:69:e0:4b:2f:e7:18:d9:9b:41:fd:98:67:1c:7c:f7:20:cf:
         ec:30:6c:eb:95:c2:c4:65:53:6e:01:86:f3:14:81:0a:13:dd:
         09:b5:9e:69:62:98:ed:f8:be:42:47:ab:4e:75:87:dc:65:a5:
         d8:c4:55:ea:62:a7:ed:7b:e8:8e:74:b7:af:d1:c4:40:5f:11:
         1b:57:5c:f4:5d:fc:19:e6:56:95:5d:b3:eb:d8:27:ef:d0:e5:
         4f:1a:ee:6b:b4:c1:98:58:6d:91:7c:b5:48:7e:97:1d:53:f7:
         92:5a:26:9b:94:3d:27:cd:df:c3:c7:9f:45:2d:f3:28:4a:a2:
         ab:ce:a8:b2:50:a9:95:68:a9:bd:57:6f:b4:2d:9c:e6:d5:c3:
         e4:a8:99:a4:6d:be:af:03:21:91:8f:ea:a7:14:e7:8b:9e:45:
         72:73:57:78:f8:1e:27:f7:17:13:75:23:69:8c:4c:5c:1b:c7:
         e6:d9:09:51:9d:d6:bc:56:be:70:c7:8e:12:61:e1:83:66:24:
         d0:59:a1:62:3c:e7:25:9c:8d:19:f8:b6:6c:a5:92:bc:95:3f:
         b6:62:0a:a3:3f:25:0d:bd:c2:65:69:4b:4c:58:71:d9:61:50:
         6b:a9:9e:90:b6:da:40:58:f1:34:d0:d8:d2:00:b1:ea:27:b2:
         97:7d:3f:11
```