TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Masaki Ihara 176154IDCR

SERVICE MESH SECURITY IN MICROSERVICES ARCHITECTURE

Diploma Thesis

Supervisor: Mohammad Tariq Meeran
Doctor of Philosophy

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Masaki Ihara 176154IDCR

TEENUSEVÕRGU TURVALISUS MIKROTEENUSTE ARHITEKTUURIS

Diplomitöö

Juhendaja: Mohammad Tariq Meeran
Doctor of Philosophy

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Masaki Ihara

17.05.2021

Abstract

Keywords: Microservices Architecture Security, Service Mesh

The microservices architecture has become increasingly popular in the IT industry; however, securing interservice traffic in a microservices-based application is a challenging task. There is a common knowledge of security measures with the microservices architecture, such as tokenization of the user context, and encryption of the traffic. This thesis aims to determine how such security measures can be practically applied to a microservices-based application. Specifically, it investigates how the service mesh pattern—an abstract infrastructure layer for a distributed application network—can be utilised to secure the application.

This research was conducted using a qualitative approach with a case study. An open-source project was analysed to identify security risks reside in a real-world project. Afterwards, the service mesh pattern was adapted to the project as a mitigation to identified issues, followed by discussions of its practicality.

The results suggest that the service mesh is a suitable solution to mitigate common security risks in the microservices architecture; however, it is not preferable for smaller scale projects with limited budgets because it is a resource-expensive solution.

This thesis is written in English and is 40 pages long, containing 6 chapters and 22 figures.

List of abbreviations and terms

ACL	Access Control List
API	Application Programming Interface
AWS	Amazon Web Service
CI/CD	Continuous Integration and Continuous Delivery
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDC	International Data Corporation
IEEE	Infrastructure as a service
Iaas	Institute of Electrical and Electronics Engineers
JWT	Json Web Token
mTLS	Mutual Transport Layer Security
NIST	National Institute of Standards and Technology
REST	Representational State Transfer
SOA	Service-oriented architecture
Saas	Software as a service
TCP	Transmission Control Protocol
TLS	Transport Layer Security

Table of contents

List of figures

1 Introduction

Software designed to be self-contained, in other words, one large chunk of system design, is known as monolithic architecture [1]. Many software applications originate from the monolithic approach because it is the simplest way to initiate a project. However, soon after the project's size grows, numerous problems can arise. Examples include scalability issues, the complexity of the codebase, and time-consuming continuous integration and continuous delivery (CI/CD), to name a few [2]. To overcome such difficulties, service-oriented architecture (SOA) was actively invested in by enterprises over the past decade. The SOA essentially decomposes a monolithic system into smaller sub-systems. It has been proven that SOA provides superior scalability and flexibility than monolithic design. Yet, it did not achieve the expected agility due to its complexity and the monolithic nature of the services built on the SOA platforms [3]. As an alternative approach and a successor, microservice architecture began to attract the IT industry [3].

1.1 Background of the Study

Microservice architecture has been rapidly increasing in popularity since the beginning of 2014, when the availability of technologies aligned with the requirements of its resource-expensive system design. The advent of technological solutions, such as Docker, Infrastructure as a Service (Iaas), and Software as a Service (Saas), significantly contributed to the adaption of the architecture in the industry [3].

Enterprises such as Netflix and Amazon have been playing important roles in this trend. As pioneers and evangelists, the so-called tech giants have taken advantage of the new architectural pattern and demonstrated the successful adaption of microservices practices. Moreover, they shared their expertise and made their tools publicly available. The practical insights they provided have increased the confidence of adaption among tech communities. Additionally, technology vendors have been influencing the community by encouraging the adaption at conferences, blogs, and any other forms of media [3].

Due to its potential and demands, microservice architecture is becoming indispensable in software development. Moreover, the amount of architecture adaption is predicted to continue growing. According to “Worldwide IT Industry 2019 Predictions” published by the International Data Corporation (IDC), the microservices architectures will be featured with 90% of all new applications by 2022. The adaption of microservices approach provides better aflexibility to design and develop the application [4].

1.2 Problem Statement and Research Questions

1.2.1 Problem Statement

The microservices architecture is generally a widely distributed system with more traffic to monitor and a larger attack surface [5]. A common enterprise application might involve dozens of microservices that are frequently communicating with one another. The loosely coupled nature of the system design makes it difficult to secure and manage internal communication channels.

Acknowledging the security risks of the microservices architecture is as important as understanding all its benefits. Network security is one of the most critical aspects of the microservices architecture. Due to the short history of the microservices architecture, we relatively lack the expertise to secure traffic within our own distributed systems.

In particular, the importance of securing communication between microservices is often overlooked and underestimated [6]. One reasoning might be that most microservices reside within private networks, and it is less intuitive to consider security risks in-house. As the number of microservices increases, so too does the complexity of traffic management, followed by the likelihood of introducing security risks.

1.2.2 Research Questions

Based on the problem statement, the main research question (RQ) is phrased as follows:

How can interservice communications be efficiently secured in microservices architecture?

To answer this question in a more structured manner, the main research question is divided into two sub-questions:

- **RQ-1: What are the potential security risks of interservice communications?**
- **RQ-2: How can service-to-service traffic be efficiently secured in a distributed polyglot system?**

As an outcome, this work will provide an overview of the security risks of service-to-service communication. Additionally, it introduces the service mesh approach as an option to mitigate the security risks.

1.3 Outline of the Thesis

This thesis is organised into chapters. Chapter 1 provides an introduction to the topic, presents the problem statement, and defines the goal of the thesis. Following this, Chapter 2 provides a literature review, describing the backbone of the microservices architecture and its security challenges, in addition to the concept of the service mesh pattern. Then, Chapter 3 describes the methodology used during the research process and identifies the assessment criteria and an open-source project to be analysed. In Chapter 4, the assessment results from a case study are presented. Afterwards, Chapter 5 discusses possible countermeasures with the service mesh. Chapter 6 concludes the thesis, while Chapter 7 discusses possible further improvements.

2 Literature Review

A thorough analysis of the service mesh security was conducted during the research. To acquire insights into the research topic, various sources were analysed. For gathering information regarding the fundamentals and security challenges of the microservices architecture, technical writings such as “Microservice Security in Action” from Manning Publications and “Microservices: The Journey So Far and Challenges Ahead” issued by the IEEE were studied. Additionally, publications focussing on the service mesh technology, such as “Building Secure Microservices-based Applications Using Service-Mesh Architecture” provided by NIST (*National Institute of Standards and Technology*) and “Istio in Action” from Manning Publications, were reviewed.

The following sections introduce and explain the fundamental concepts and approaches of this thesis. First, Section 2.1 offers a high-level introduction to microservices architecture and provides the reader with a basic understanding of the system design. Following this, the challenges associated with securing service-to-service traffic are described in Section 2.2. Then, Section 2.3 introduces the concept of the service mesh, which is followed by a description of the detected gap in Section 2.4.

2.1 Microservice Architecture

The following sections provide a high-level overview of microservices architecture to give the reader a basic understanding of the architecture and its purpose.

2.1.1 Overview

Microservices are the latest trend in software service design, development, and delivery [7]. Microservices design is a composed approach to software and systems architecture which based on the concept of modularisation but emphasises technical boundaries. Each module, also referred to as a microservice, is implemented and operated as a small yet independent system, providing access to its internal logic and data through well-defined network interfaces, such as REST API [8]. Improved software agility could be achieved

using this approach because each microservice becomes an independent unit of development, deployment, operation, versioning, and scaling [9].

2.1.2 Comparison with Monolithic Architecture

Monolith is an ancient word referring to a large block of stone [10]. The concept of monolithic software involves different components of an application being combined into a single program on a single platform [11]. Normally, a monolithic application consists of a database, client-side user interface, and server-side application [12]. All the software's components are unified, and all its functions are managed in a single location. This approach supports simple development and deployment. Hence, this is the most affordable option for starting projects, particularly when the project is run by a small team [9].

2.1.3 Polyglot Design

In microservice deployment, services interact with one another over the network, relying on each service's interface. One benefit of introducing microservices architecture is the flexibility for the choice of programming languages and technology stacks for implementation. In a multi-team environment, in which a set of microservices are developed by each team, the teams have the freedom to select the most suitable technology stack for their respective requirements [6]. This architecture, which promotes different components in a system to select the technology stack that is best for itself, is known as the polyglot architecture [6].

2.1.4 Traffic Types in Microservices

Microservice architecture generally involves two types of communication traffic, namely North/South traffic and West/East traffic [6].

North/South Traffic

This is a type of traffic that moves in and out of a private network [13]. Traffic from the client to the server, such as a web browser attempting to fetch data from an application programming interface (API) server, is an example of North/South traffic.

West/East Traffic

As opposed to North/South traffic, West/East traffic is a type of traffic between one server and another within a private network; therefore, service-to-service communication is an example of West/East traffic [13].

2.2 Challenges with Interservice Communication Security

The following sections describe some of the security challenges with service-to-service communication.

2.2.1 Distributed Security Screening May Degrade Performance

More microservices results in more interconnections among microservices and more traffic to be protected [14]. Unlike in a monolithic application, independent security screening must be executed for each microservice. Having multiple security screenings at the entry point of each microservice might appear to be redundant from the perspective of a monolithic application, which performs security screening once, after which the request is dispatched to the corresponding component. The distributed security checks that occur repeatedly on each service interaction might result in latency and considerably degrade the performance of the system [6].

A workaround to avoid repetitive security checks might be to simply trust the network. However, trust-the-network has been acknowledged as an antipattern in recent years, and the industry is shifting towards zero-trust networking principles [6]. Any microservices security design must consider overall performance and take precautions to address any drawbacks [6].

2.2.2 Bootstrapping Trust Among Microservices is Difficult

Today, large-scale microservice deployments with hundreds of services are no longer a surprise. For example, Monzo, an online bank based in the United Kingdom, runs more than 1,600 microservices on AWS [15]. Managing microservices deployment with even dozens of services would be challenging without automation. Each microservice should be provisioned with a certificate. This certificate is used for authentication during service-to-service interactions [16]. However, microservices can come and go dynamically, which makes managing the certificate difficult.

2.2.3 Sharing User Context is More Difficult in a Distributed System

All microservices must be treated as non-trustworthy [14]. Internal components share the same web session in a monolithic application, and anything related to the requesting party is retrieved from it [6]. Meanwhile, achieving the same result in microservices architecture requires greater effort. Nothing, or a very limited set of resources, could be shared among microservices, leading to a situation in which the user context must be passed explicitly from one microservice to another. The challenge is to build trust between two microservices such that the receiving microservice obtains the user context sent from the other [6]. Thus, the integrity of the passed user context must be verified to prevent deliberate modification [6].

2.2.4 Polyglot Architecture Demands More Security Expertise

Security is more challenging with the polyglot architecture. Since different teams use different technology stacks for development, each team must have its own security expertise. As such, each team is responsible for security practices, guidelines, and integration with existing tools and systems [6].

2.3 Service Mesh Pattern

The following sections describe a service mesh, which is a method to control how different parts of an application share data with one another. It is a dedicated infrastructure layer built directly into an application [17].

2.3.1 Why Service Mesh

Due to the security challenges of microservices-based applications stated in the previous section, the infrastructure that supports the application and the infrastructure's associated service should be tightly coordinated [14]. A service mesh is such a dedicated infrastructure layer.

2.3.2 What is Service Mesh

The term service mesh is used to describe the network of distributed microservices systems and the interactions between them. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring. In addition, a service mesh

commonly has more complex operational requirements, such as A/B testing, canary rollouts, rate limiting, access control, and end-to-end authentication. [18]

2.3.3 Istio

An open-source project, Istio, might be the most popular service mesh implementation. Istio service mesh support is added to services by deploying a special sidecar proxy throughout the application environment that intercepts all network traffic between microservices. Then, the traffic can be configured and managed by Istio, utilising its control plane functionality as illustrated in **Figure 1** [18].

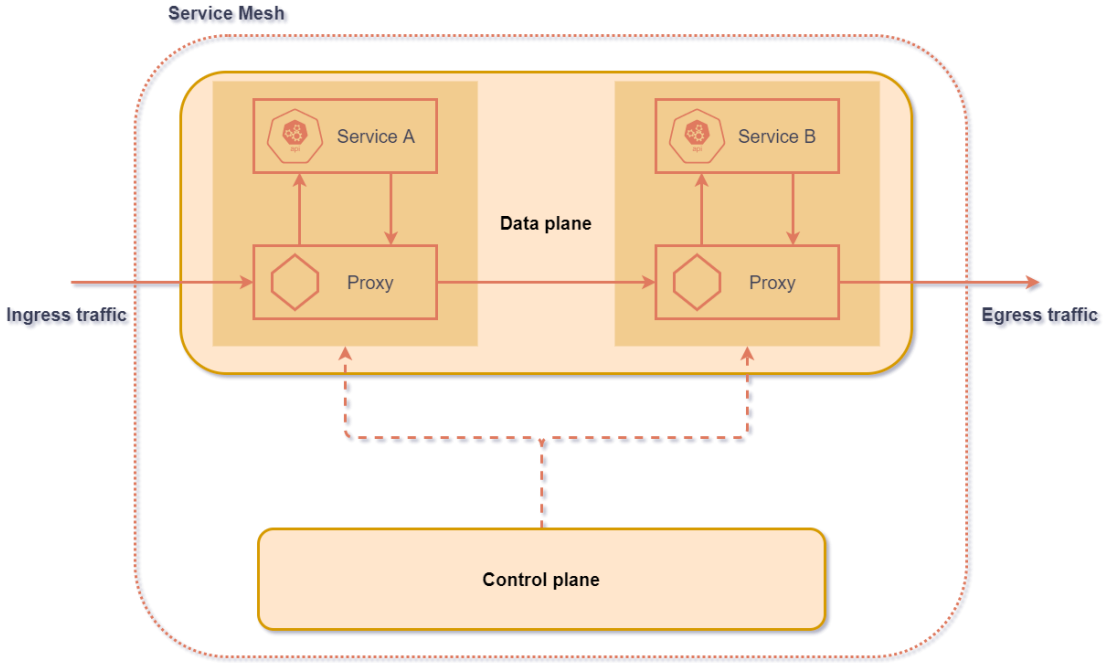

Figure 1. Istio Service Mesh Design Diagram [18]

2.3.4 Architecture

An Istio service mesh is logically split into two planes, as follows.

Data plane

The data plane is composed of a set of intelligent proxies deployed as sidecars. The sidecar proxies mediate and control all network traffic between microservices [19].

Control plane

The control plane manages and configures the proxies to route traffic [19].

2.3.5 Core Features

Istio can provide the following capabilities uniformly across a network of services:

- Traffic management
- Security
- Observability

Traffic Management

The flow of traffic and API calls between services can be controlled by rules configuration and traffic routing. For example, circuit breakers, timeout and retry, and A/B testing can be configured via traffic management [20].

Security

Important aspects of any microservices application, such as the management of authentication and authorisation or encryption of pod-to-pod communication, can be configured with the security capability [21].

Observability

The service mesh provides robust tracing, monitoring, and logging features, which offer deep insights into deployed service mesh [22].

2.3.6 Mutual TLS

A mutual TLS, also known as mutual authentication or mTLS, is a security process in which entities authenticate one another before actual communication occurs [23]. In the context of Istio service mesh, keys and certificates for mTLS are automatically installed in all sidecar containers by the control plane [24]. Therefore, communication between sidecars can be encrypted with mTLS, while no changes are required for application.

2.4 Detecting a Gap

This thesis aims to produce two main contributions.

First, it aims to provide the analysis results of an open-source project that assesses security risks that reside in a real-world project. This verifies theoretical issues and factual risks. Second, it aims to implement the service mesh pattern to the analysed project to evaluate the security improvements.

While the reviewed literature focussed on the conceptional solution with examples or limited implementations, this thesis practices the theoretical knowledge using a non-fictional project.

3 Methodology

To achieve the goals of this thesis, research was conducted using a qualitative approach with a case study.

To answer research question RQ-1—what are the potential security risks of interservice communications?—an open-source project was analysed to obtain a better understanding of the research problem.

Based on the result of RQ-1 and the risk analysis, a service mesh pattern was applied to the analysed project as part of RQ-2—how can service-to-service traffic be secured in a polyglot system?

Section 3.1 provides a brief introduction to the subject of the case study. Then, the assessment criteria are presented in Section 3.2. Following this, Section 3.3 explains the environment setup used during the research.

3.1 Selected Application

For the rational case study, the open-source project ‘microservice-app-example’, which appears to be popular, with more than 1.3k stars on GitHub, was selected. The application simulates real-world system design and demonstrates the practical use of polyglot microservices architecture. The application is composed of a web user interface plus four microservices in different technologies, namely Java, Python, Node.js, and Go.

3.2 Assessment Criteria

To evaluate the security risks of the open-source microservices-based application, the following assessment criteria were identified based on the reviewed literature: Microservices Security in Action [6].

- Distributed security screening

- Traffic encryption
- Secure user-context sharing
- Polyglot system design

As a result of this assessment, it was expected to identify the relevant security risks that reside in public projects.

3.3 Environment Setup

The assessment was performed using the following components and tools:

- Windows 10 PC (Version 10.0.19041 Build 19041)
- Docker (v20.10.5)
- Docker Compose (v1.29.0)
- Kubernetes (v1.19.7)
- kubectl (v 1.20.6)
- Ksniff (v1.6.0)
- Wireshark (v3.4.4)

3.3.1 Docker Engine Setup

The application needs Docker images to be built to deploy to a kubernetes cluster. Building a Docker image requires the Docker Engine to be installed. Since the assessment was performed on a Windows machine, the fastest and easiest way to get started with Docker on Windows [25] was to install Docker Desktop for Windows, as presented in **Figure 2.**

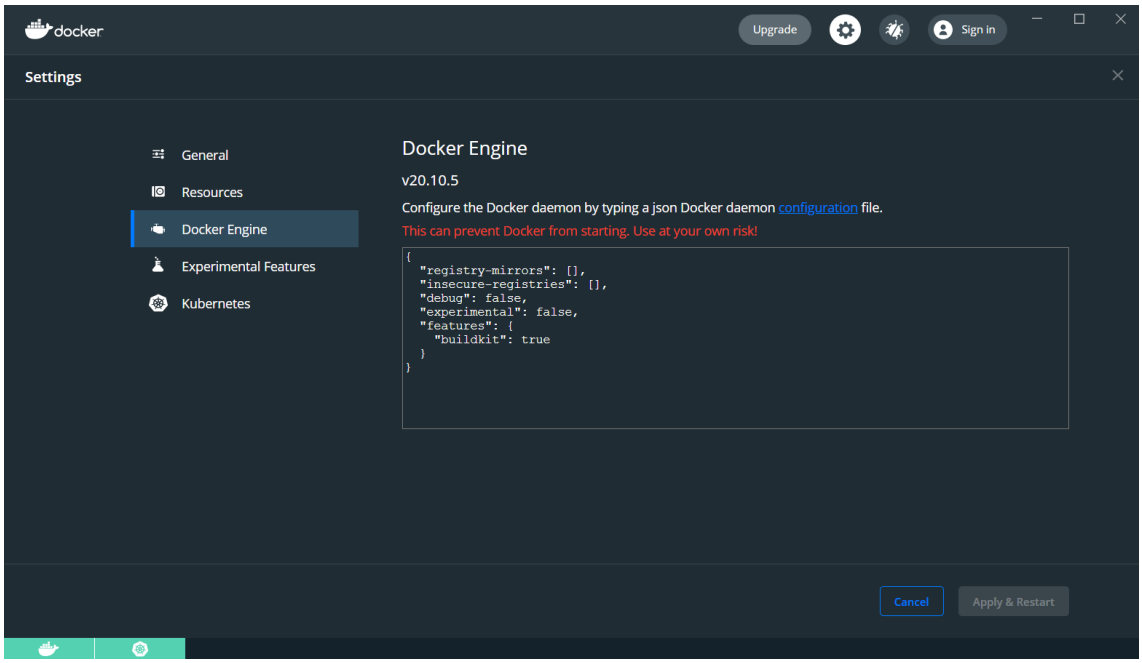

Figure 2. Confirmation of Docker Engine Installation

Together with the Docker Engine, Docker Compose was installed (**Figure 3**). Docker Compose might require a separate installation in the case of Linux environments.

```
C:\Users\mxxxx>docker-compose version
docker-compose version 1.29.0, build 07737305
docker-py version: 5.0.0
CPython version: 3.9.0
OpenSSL version: OpenSSL 1.1.1g  21 Apr 2020
```

Figure 3. Confirmation of Docker Compose Installation

3.3.2 Kubernetes Server and Client Setup

To perform an assessment with a context of runtime, it was required to deploy Docker images built from the application onto a kubernetes cluster. While there are many options for kubernetes implementation and providers [26], the Kubernetes single-node cluster support from Docker Desktop for Windows was selected for the sake of simplicity.

Kubernetes support is disabled by default; therefore, the feature was manually enabled, as presented in **Figure 4**.

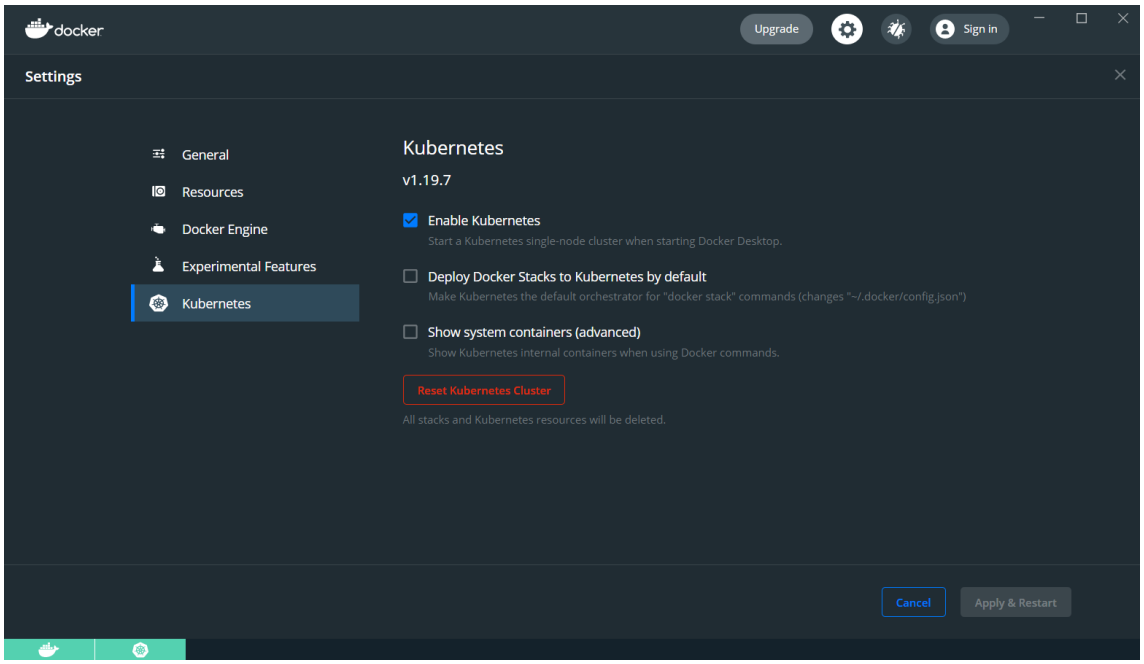

Figure 4. Confirmation of Kubernetes Installation

When the Kubernetes feature is enabled, the Kubernetes server and client become available (**Figure 5**).

```
C:\Users\mxxxx>kubectl version
Client Version: version.Info{Major:"1", Minor:"20", GitVersion:"v1.20.5", GitCommit:"6b1d87acf3c8253c123756b9e
61dac642678305f", GitTreeState:"clean", BuildDate:"2021-03-18T01:10:43Z", GoVersion:"go1.15.8", Compiler:"gc",
 Platform:"windows/amd64"}
Server Version: version.Info{Major:"1", Minor:"19", GitVersion:"v1.19.7", GitCommit:"1dd5338295409edcfff11505e
7bb246f0d325d15", GitTreeState:"clean", BuildDate:"2021-01-13T13:15:20Z", GoVersion:"go1.15.5", Compiler:"gc",
 Platform:"linux/amd64"}
```

Figure 5. Kubernetes Server and Client Versions

3.3.3 Application Setup

Once Docker and Kubernetes were ready, the build steps described in a README form within the application were followed [27]. As a result, five Docker images were built for the application, as illustrated in **Figure 6**.

```
C:\Users\mxxxx\sources\Personal\microservice-app-example> docker images
REPOSITORY                          TAG                 IMAGE ID       CREATED          SIZE
frontend                            latest              dc1f03b0134d   17 minutes ago   257MB
todos-api                           latest              5ca65cc17f99   18 minutes ago   88.1MB
log-message-processor               latest              0a8e38a4c80d   18 minutes ago   248MB
auth-api                            latest              ee638d3aef04   18 minutes ago   372MB
users-api                           latest              57d6d03942aa   19 minutes ago   267MB
```

Figure 6. List of Docker Images Built

Afterwards, the built Docker images were deployed to the Kubernetes cluster (**Figure 7**).

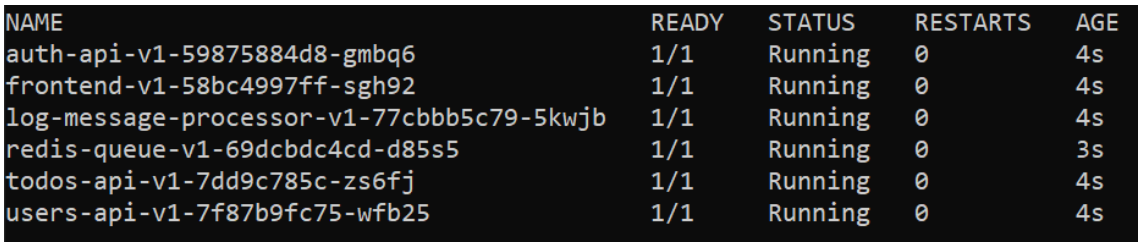

Figure 7. Deployed Pods

3.3.4 Packet Analyser Setup

Analysing packages between microservices was somewhat intricate, as the network traffic never leaves the Kubernetes cluster in a single node setup. Therefore, it was necessary to append an additional pod to intercept service-to-service traffic, as displayed in **Figure 8**. Ksniff is a suitable tool to perform such an operation; it utilises tcpdump to capture traffic on any pod within the network [28], [29].

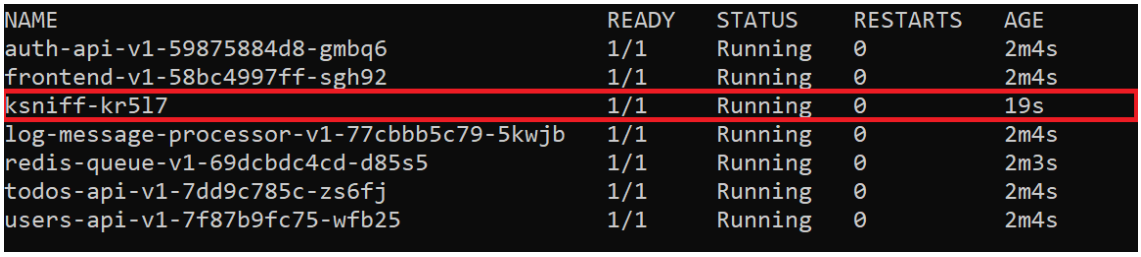

Figure 8. Ksniff Pod

Once the Ksniff pod begins capturing a target pod, dumped traffic packets can be analysed using Wireshark, as displayed in **Figure 9**.

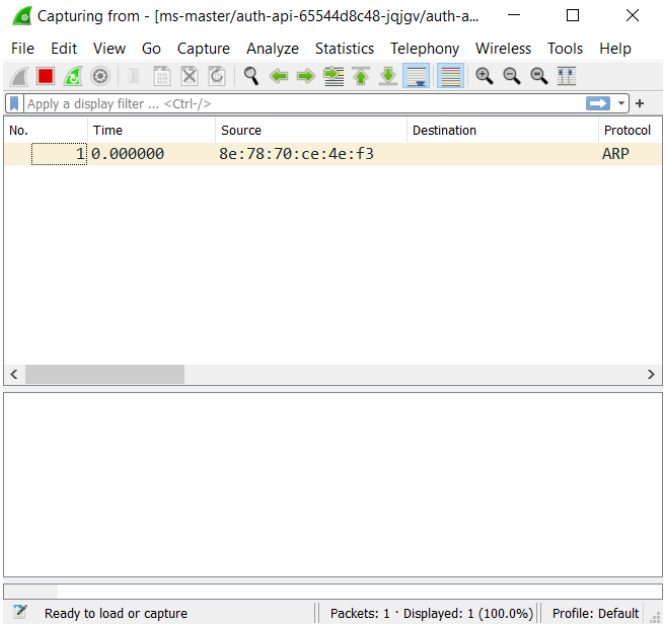

Figure 9. Wireshark Capturing Packet via Ksniff

3.3.5 Summary

After completing all the steps mentioned above, the application was running in a single node cluster (**Figure 10**), and interservice communication packets could be monitored and analysed.

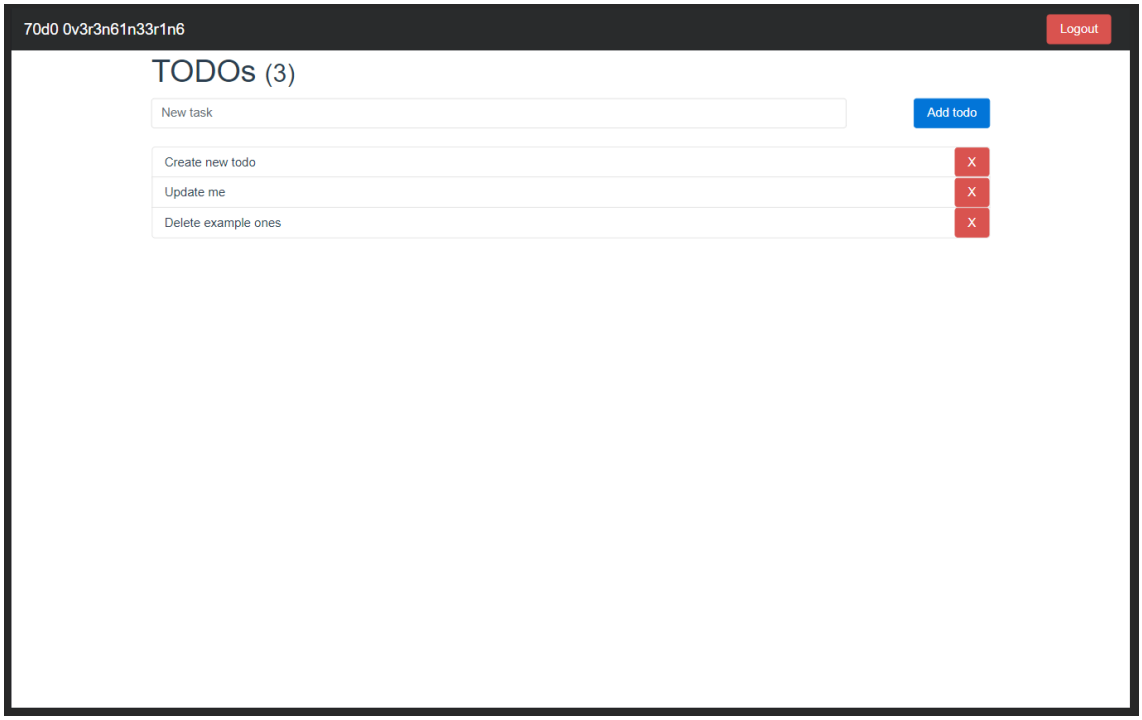

Figure 10. Application Web Page

4 Risk Assessment Results

To evaluate potential security issues, a risk assessment was conducted. This chapter explains the issues identified during the assessment.

The following sections present the results of the performed risk assessment. First, Section 4.1 introduces to the application topology. Second, Section 4.2 describes the identified issue with the unencrypted traffic. Third, the risks of lacking access control policy are described at Section 4.3. Following this, the problem of the dependency management in the polyglot system design is described in Section 4.4.

4.1 Application Topology

Figure 11 illustrates each component of the application and its interaction.

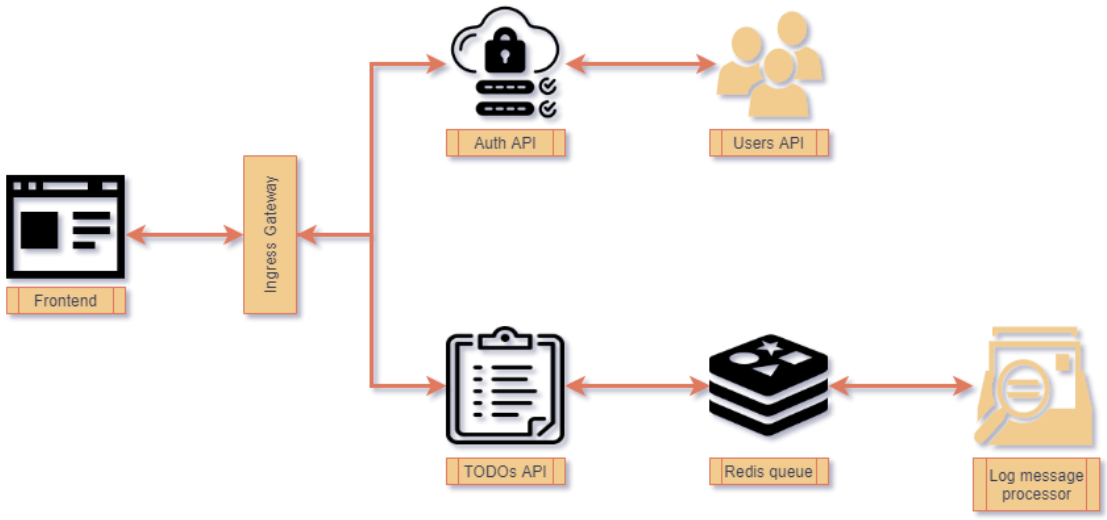

Figure 11. Application Topology (Inspired by [27])

4.2 Unencrypted Interservice Traffic

Having observed traffic between the Auth API and Users API, the request payload could be captured in plain text because the request was transported over HTTP (**Figure 12**).

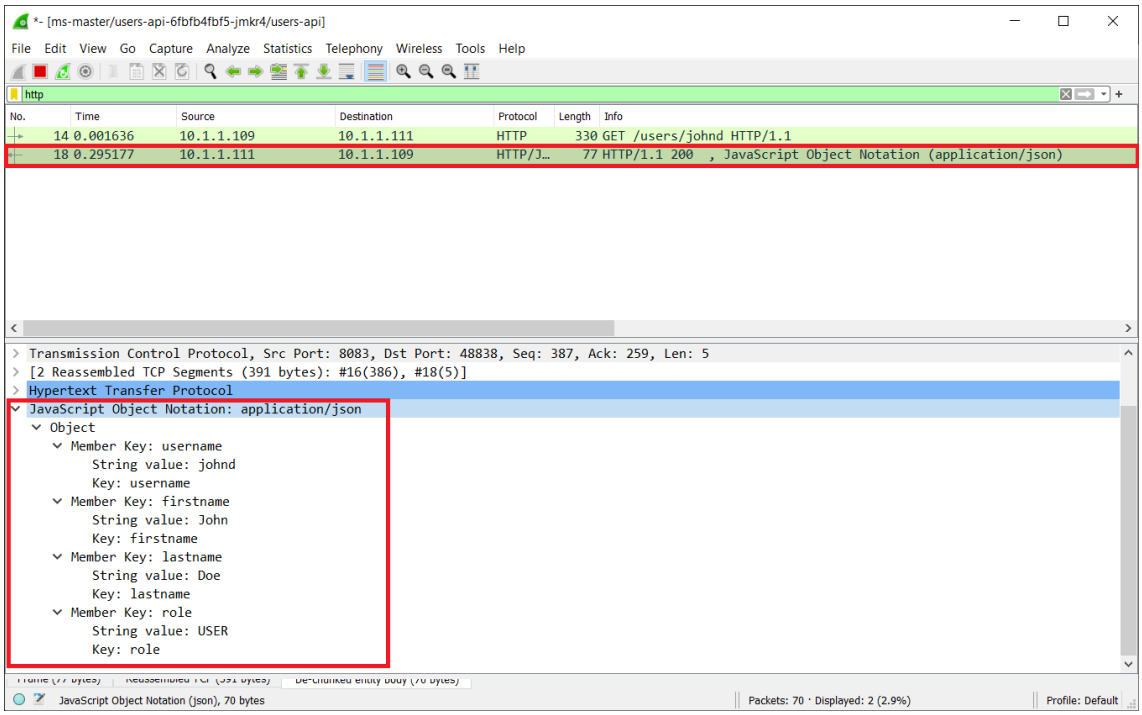

Figure 12. Captured Packet Between Auth API and Users API

Non-encrypted traffic could be acceptable if the traffic never leaves the cluster node and is processed only internally; however, this is less likely the case in modern days. Kubernetes would be running with a multi-cluster setup for better availability and scalability in a typical production environment, which might result in HTTP traffic running from one node to another over an external network from the perspective of one cluster node.

Figure 13 illustrates an example of a multi-cluster setup. The user information captured in an earlier step would be observable in the overlay network in the case of Pod 1 (Auth API) performing the same request to Pod 4 (Users API).

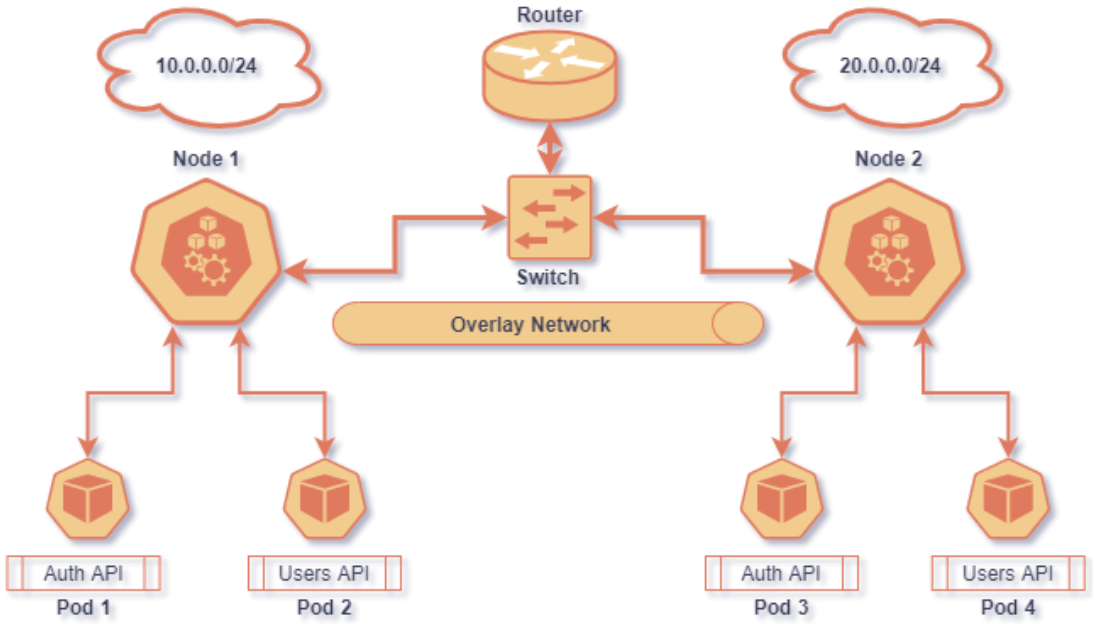

Figure 13. Example of Multi-Cluster Setup Diagram

4.3 No Presence of Access Control Policy

Another issue discovered was that the lack of an access control list for the API endpoints. Having no access control policy could result in a situation such that anybody with access to the network can make requests to any API endpoint. The current state of the application unnecessarily leaves room for security risks.

To demonstrate the security risk, a pod performing the role of unauthorised access was deployed to determine how it can interact with other services within the network. **Figure 14** highlights the pod with unauthorised access.

```
NAME                                    READY   STATUS    RESTARTS   AGE
auth-api-v1-59875884d8-gmbq6            1/1     Running   0          10m
frontend-v1-58bc4997ff-sgh92            1/1     Running   0          10m
ksniff-kr5l7                            1/1     Running   0          8m25s
log-message-processor-v1-77cbbb5c79-5kwjb   1/1     Running   0          10m
redis-queue-v1-69dcbdc4cd-d85s5         1/1     Running   0          10m
sleep-557747455f-rzr2d                  1/1     Running   0          4s
todos-api-v1-7dd9c785c-zs6fj            1/1     Running   0          10m
users-api-v1-7f87b9fc75-wfb25           1/1     Running   0          10m
```

Figure 14. Additional Pod Performing Unauthorised Access

With the pod deployed, the following bash script (**Figure 15**) was executed to exercise unauthorised HTTP requests to Auth API and Users API. The result of the script is displayed in **Figure 16**.

```
#!/bin/bash

echo "sleep pod is sending request to auth-api"
kubectl exec "$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})" -c sleep \
  -- curl -X POST -H "Content-Type: application/json" \
  -d '{"username": "johnd", "password": "foo"}' \
  auth-api:8081/login -s -o /dev/null -w "%{http_code}\n"

echo "sleep pod is sending request to users-api"
kubectl exec "$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})" -c sleep \
-- curl -X GET -H "Content-Type: application/json" -H "Authorization: Bearer some.invalid.token" \
users-api:8083/users/johnd -s -o /dev/null -w "%{http_code}\n"
```

Figure 15. Bash Scripts to Exercise Unauthorised HTTP Request

```
sleep pod is sending request to auth-api
200
sleep pod is sending request to users-api
500
```

Figure 16. Result of the Bash Script

The result (**Figure 16**) indicates that the unauthorised pod could send HTTP requests to the API endpoints successfully. Note that the HTTP response status code of 500 (Internal Server Error) from the Users API reveals that the request reached the application; however, an error occurred due to the invalid json web token (JWT).

4.4 Different Zipkin Clients are Managed

In the analysed project, Zipkin is used as a distributed tracing system. It is deployed and managed the same as other microservices. Due to the polyglot design, each client microservice installs the Zipkin client library to submit telemetry to the central server. While having a distributed tracing system is an advantage to maintaining an application, managing the client library from the different services and technologies is overhead and might create security risks caused by the dependency of certain client libraries.

5 Discussion

This chapter discusses the mitigation of the identified security risks in Chapter 4. Section 5.1 introduces to the mTLS to secure the service-to-service traffic. Following this, Section 5.2 discusses the service mesh pattern in details as a countermeasure to each of the identified risks.

5.1 Internal Traffic with Mutual TLS

Considering the same example as in the risk assessment, the service-to-service request between Auth API and Users API was carried over HTTP. In the ideal scenario, all the traffic flowing in the network should be encrypted even if the network is supposedly private.

A good method to overcome the issue is to apply an mTLS pattern. Having bi-directional traffic encryption ensures a secure East/West channel. However, the dynamic nature of the microservices system makes it difficult to achieve the mTLS strategy.

Commonly, there are multiple instances of a service managed in a microservice system. From the perspective of an application, it is difficult to know how many replicas of the instance exist and which of the instances of other services it is communicating with. Moreover, the polyglot aspect of system design introduces another level of difficulty in managing certificates for each service.

5.1.1 Utilise Service Mesh for mTLS

The service mesh design, which controls how different parts of an application share data with one another [17], is a practical approach for overcoming the issue with traffic encryption. Having another layer of abstraction directly in front of the pod allowed us to solve such a security issue without making changes to the API service, itself.

5.2 Assessment with Service Mesh

5.2.1 Application Topology with Istio Service Mesh

Once the service mesh is deployed, every pod is put behind a proxy service, as illustrated in **Figure 17**. Now, each traffic is routed via the originating proxy service to the destination proxy service.

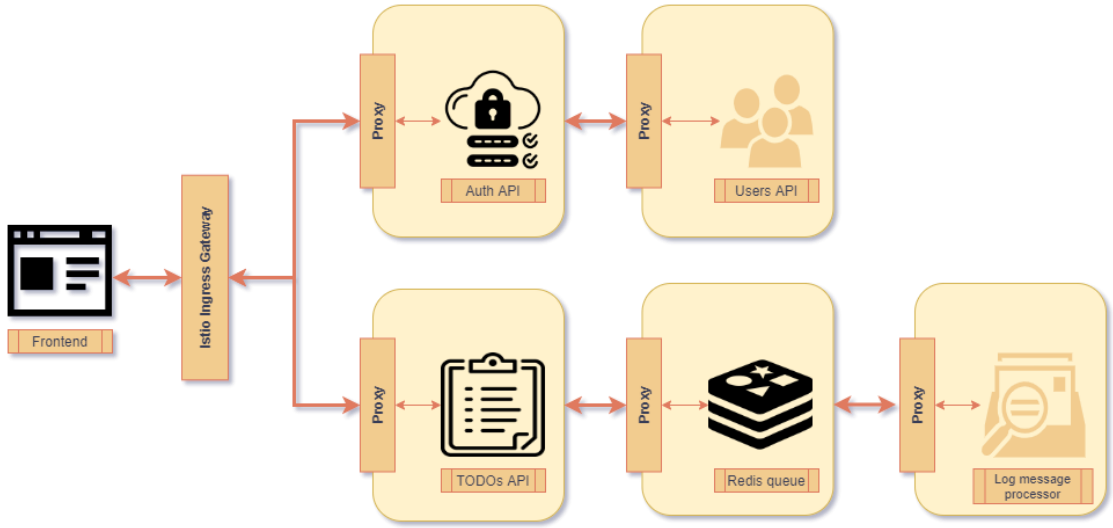

Figure 17. Application Topology with Istio Service Mesh

5.2.2 Service-to-Service Traffic with mTLS

When a service is covered by proxy, mTLS can be applied to traffic without modifying the application logic. In the case of Istio, mTLS can be strictly applied with a control plane configuration.

The following configuration (**Figure 18**) enforces a control plane to perform a mesh-wide peer authentication with mTLS.

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"
  namespace: "istio-system"
spec:
  mtls:
    mode: STRICT
```

Figure 18. Istio Strict mTLS Configuration

Now, each traffic is encrypted with mTLS, and it should not be observed in the middle of traffic. More closely examining the captured packet (**Figure 19**), the same request is still carried over HTTP in plain text; however, it can be observed that the source and destination IP addresses are both 127.0.0.1, as opposed to the request analysed in the assessment chapter, which used the actual IP address of a pod.

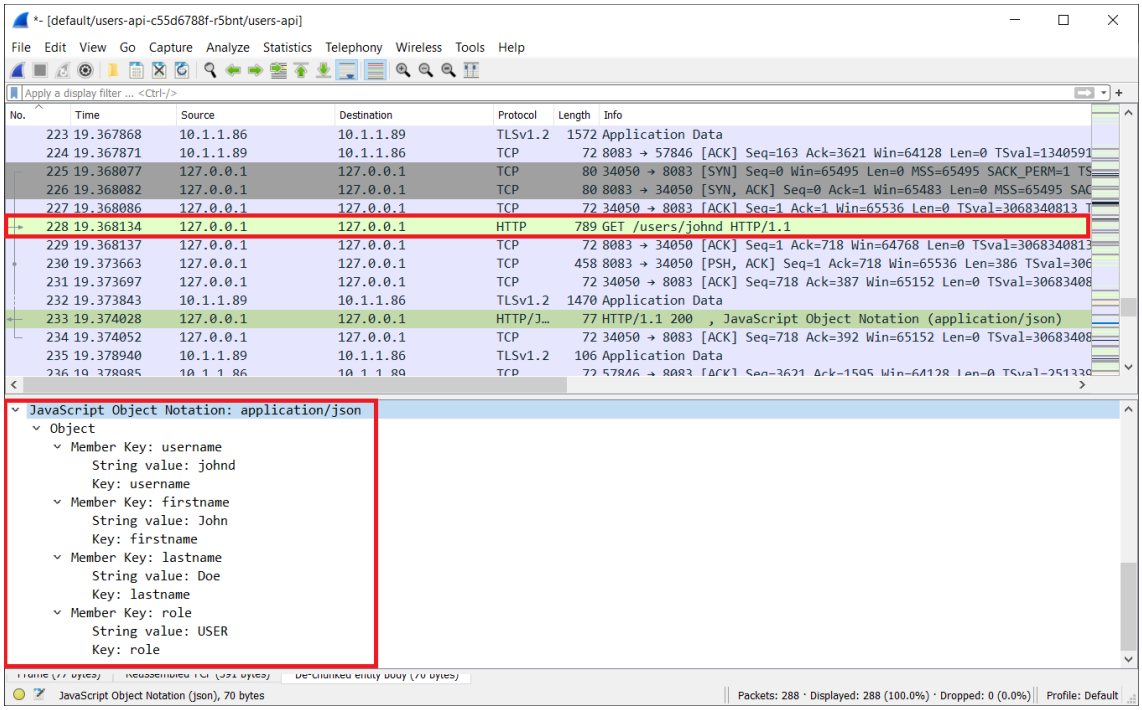

Figure 19. Captured Packet Between Auth API and Users API with Service Mesh

This indicates that the request was returned to the front-line proxy service running within the same container and thus sharing the IP address of 127.0.0.1 (localhost). Additionally, it can be observed that prior to the returned HTTP response, a TLS connection was established between the actual IP addresses of the pods.

5.2.3 Service-Level Authorisation Policy

The data plane forwards the network topology of the system to the control plane. Therefore, the control plane can be configured to manage authorisation policies.

The configurations below (**Figure 20, Figure 21**) serve as an example of access control policies. In a short description, the Auth API's policy (**Figure 20**) explicitly allows only login requests via Istio's Ingres Gateway, while the Users API's policy (**Figure 21**) explicitly allows the Auth API to send only GET requests to the '/users' endpoint.

```yaml
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "auth-api-policy"
  namespace: "default"
spec:
  selector:
    matchLabels:
      app: auth-api
  action: ALLOW
  rules:
  - from:
      - source:
          principals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-service
              -account"]
    to:
      - operation:
          methods: ["POST"]
          paths: ["/login"]
```

Figure 20 Authorisation Policy for Auth API

```yaml
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "users-api-policy"
  namespace: "default"
spec:
  selector:
    matchLabels:
      app: users-api
  action: ALLOW
  rules:
  - from:
      - source:
          principals: ["cluster.local/ns/default/sa/auth-api"]
    to:
      - operation:
          methods: ["GET"]
          paths: ["/users"]
```

Figure 21 Authorisation Policy for Users API

The whitelisting approach ensures that only the minimum required API endpoint is exposed to only relevant API services. Thus, running the same bash script as in the assessment step should not succeed with authorisation policies enabled.

```
sleep pod is sending request to auth-api
403
sleep pod is sending request to users-api
403
```

Figure 22. Result of the Bash Script with Authorisation Policy

As presented in **Figure 22**, the unauthorised pod now receives the HTTP response status code of 403 (Forbidden).

5.2.4 Distributed Tracing

As a proxy deployed to each microservice essentially acts as a client to the central system, it is easier to provide the capability of distributed tracing. Istio comes with Kiali, its de facto distributed tracing system, which can replace Zipkin. With a service mesh deployed, the overhead of managing different Zipkin clients caused by polyglot design could be eliminated.

5.2.5 Downsides of the Service Mesh

In the previous sections, the benefits of the service mesh pattern were discussed. Although having another infrastructure layer provides conveniences, there are certain drawbacks when considering its application.

Immaturity

The service mesh is a fairly new concept, which began gaining attention in the past few years. Version 1.0 of Istio was released in the middle of 2018 [30]. Due to the short history of the solution, documentations are often outdated, and the amount of available resources is still low [31].

Running Cost

Operating a service mesh alongside the containers that run application services requires additional resources. It could strain the budget, as it adds compute overhead to run [32].

6 Conclusion and Future Work

This chapter concludes this thesis, summarises the research efforts, and answers the research questions outlined in Chapter 1. Section 6.1 provides a general conclusion of the thesis, which is followed by Section 6.2 that answers each of the defined research questions independently. Afterwards, Section 6.3 describes the imitations of this thesis. Finally, Section 6.4 provides an outlook on the future work.

6.1 Conclusion

The microservices architecture has become an essential system design in the software industry over the past demi-decade. However, the concept of widely distributed systems is still relatively new, which is a concerning fact from the perspective of security. In this study, some of the potential security risks residing in real-world microservices-based applications were identified.

A case study focussing on internal traffic security was conducted with a popular open-source project to demonstrate the use of polyglot microservices design. Then, a few security concerns were discovered with the risk assessment, namely non-encrypted traffic and the lack of ALC policy. Following the risk assessment, mitigation with the service mesh as an additional security layer was discussed. Enabling mTLS and explicit access control with the service mesh provides a better security policy for service-to-service communication.

6.2 Answering the Research Questions

The main research question of this thesis is as follows: How can interservice communications be efficiently secured in microservices architecture? This research question is divided into two sub-questions to answer the main question in a more

structured manner. The following sections conclude the answers to each of the sub-questions.

6.2.1 RQ-1: What are the potential security risks of interservice communications?

There are several major security risks that reside in interservice communications. In this thesis, a few were discovered from the analysed open-source project. First, it was identified that service-to-service traffic is carried over a plane HTTP request, which exposed the requested payload in an observable form. Another discovery was that the application lacks an access control policy that allows anybody in the network to interact with any services.

6.2.2 RQ-2: How can service-to-service traffic be efficiently secured in distributed polyglot system?

In microservice architecture, it is difficult to bootstrap trust among microservices and manage access control policies from the viewpoint of deployed services. Therefore, there should be a central method of controlling such security aspects for better manageability. Having a service mesh is a practical solution for providing a central point to take control over the security aspects of running services regardless of the underlying technologies or deployment complexities.

6.3 Limitations

Certain limitations should be evaluated beforehand. The service mesh pattern is still a very new concept. Therefore, the number of information sources is few, yet the documentations are often outdated due to the faster cycle of updates. In addition, operating a service mesh is not for everyone, particularly for smaller teams with limited budgets. Running a service mesh involves greater resource usage, and thus, the operational cost might be expensive.

6.4 Future Work

This thesis focusses on security measures in interservice communications with the service mesh pattern. A concept of the service mesh provides a distributed application with a wide range of supplemental features, many of which could not be covered with this thesis.

For further development, uncovered topics, such as gPRC and Requests Authentication, could be studied and practiced with a non-fictional project. In addition, the service-to-service security can be addressed in different layers of the system. While this study focussed on the service mesh layer, further investigations could be made into Kubernetes configuration, application design, and team and security policy management, to list a few examples.

References

[1] "Monolithic application." Wikipedia. 2020. [Online]. Available: https://en.wikipedia.org/wiki/Monolithic_application (accessed Jan. 26, 2020).

[2] "Microservices vs Monolith: which architecture is the best choice?" N-ix. 2018. [Online]. Available: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ (accessed Jan. 26, 2020).

[3] "5 Reasons why Microservices have become so popular in the last 2 years." LinkedIn. 2016. [Online]. Available: https://www.linkedin.com/pulse/5-reasons-why-microservices-have-become-so-popular-last-sakhuja/ (accessed Jan. 19, 2020).

[4] "IDC Unveils the Top Ten Predictions of Digital Transformation and Technologies for 2019 and Beyond in Indonesia." IDC. No date. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prAP44833919 (accessed Jan. 19, 2020).

[5] "13 Best Practices to Secure Microservices." Greek Flare. 2020. [Online]. Available: https://geekflare.com/securing-microservices/ (accessed Apr. 19, 2021).

[6] Prabath Siriwardena and Nuwan Dias, *Microservices Security in Action*. Shelter Island: Manning Publications, 2020.

[7] O. Zimmermann, "Microservices tenets: Agile approach to service development and deployment," *Computer Science - Research and Development*, vol. 32, no. 3–4, pp. 301–310, Jul. 2017, doi: 10.1007/s00450-016-0337-0.

[8] "Microservices," Martin Fowler. 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html (accessed May 08, 2021).

[9] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3. IEEE Computer Society, pp. 24–35, May 01, 2018, doi: 10.1109/MS.2018.2141039.

[10] “MONOLITH.” Cambridge Dictionary. No date. [Online]. Available: https://dictionary.cambridge.org/dictionary/english/monolith (accessed Jan. 26, 2020).

[11] Jalel T, “Monolith, SOA, Microservices, or Serverless?” LinkedIn. 2019. [Online]. Available: https://www.linkedin.com/pulse/monolith-soa-microservices-serverless-jalel-tounsi/ (accessed May 08, 2021).

[12] Chris Richardson, “Monolithic Architecture pattern.” Micro Services. 2019. [Online]. Available: https://microservices.io/patterns/monolithic.html (accessed May 08, 2021).

[13] “East/West Is the New North/South.” Dzone. 2018. [Online]. Available: https://dzone.com/articles/eastwest-is-the-new-northsouth (accessed Jan. 28, 2020).

[14] R. Chandramouli and Z. Butcher, “NIST Special Publication 800-204A Building Secure Microservices-based Applications Using Service-Mesh Architecture,” *NIST Pubs*, 2020, doi: 10.6028/NIST.SP.800-204A.

[15] “Monzo Case Study.” Amazon Web Services. 2018. [Online]. Available: https://aws.amazon.com/solutions/case-studies/monzo/ (accessed May 08, 2021).

[16] Christian E. Posta and Rinor Maloku, *Istio in Action*. Shelter Island: Manning Publications, 2021.

[17] “What’s a service mesh?” Red Hat. No date. [Online]. Available: https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh (accessed Apr. 24, 2021).

[18] “What is Istio?” Istio. No date. [Online]. Available: https://istio.io/docs/concepts/what-is-istio/ (accessed Jan. 30, 2020).

[19] “Architecture.” Istio. No date. [Online]. Available: https://istio.io/latest/docs/ops/deployment/architecture/ (accessed Apr. 25, 2021).

[20] “Traffic Management.” Istio. No date. [Online]. Available: https://istio.io/latest/docs/concepts/traffic-management/ (accessed Apr. 24, 2021).

[21] “Security.” Istio. No date. [Online]. Available: https://istio.io/latest/docs/concepts/security/ (accessed Apr. 24, 2021).

[22] “Observability.” Istio. No date. [Online]. Available: https://istio.io/latest/docs/concepts/observability/ (accessed Apr. 24, 2021).

[23] “Mutual Authentication.” Learn Akami. No date. [Online]. Available: https://learn.akamai.com/en-us/webhelp/iot/internet-of-things-over-the-air-user-

guide/GUID-21EC6B74-28C8-4CE1-980E-D5EE57AD9653.html (accessed Apr. 25, 2021).

[24] “Istioldie 1.0 / Mutual TLS Deep-Dive.” Istio. No date. [Online]. Available: https://istio.io/v1.0/docs/tasks/security/mutual-tls/ (accessed Apr. 25, 2021).

[25] “Docker Desktop for Windows.” Hub Docker. No date. [Online]. Available: https://hub.docker.com/editions/community/docker-ce-desktop-windows (accessed Apr. 22, 2021).

[26] “Partners.” Kubernetes. No date. [Online]. Available: https://kubernetes.io/partners/ (accessed Apr. 22, 2021).

[27] “elgris/microservice-app-example: Example of polyglot microservice app.” Github. No date. [Online]. Available: https://github.com/elgris/microservice-app-example (accessed Apr. 22, 2021).

[28] “TCPDUMP/LIBPCAP public repository.” TCPDUMP. No date. [Online]. Available: https://www.tcpdump.org/ (accessed Apr. 22, 2021).

[29] “eldadru/ksniff: Kubectl plugin to ease sniffing on kubernetes pods using tcpdump and wireshark.” Github. No date. [Online]. Available: https://github.com/eldadru/ksniff (accessed Apr. 22, 2021).

[30] “1.0.x Releases.” Istio. 2018. [Online]. Available: https://istio.io/latest/news/releases/1.0.x/ (accessed Apr. 29, 2021).

[31] “What is Istio service mesh and when to use it?” Objectivity Blog. 2019. [Online]. Available: https://www.objectivity.co.uk/blog/what-is-istio-service-mesh-and-when-to-use-it/ (accessed Apr. 29, 2021).

[32] “Evaluate key service mesh benefits and architecture limitations.” Search it Operations. 2019. [Online]. Available: https://searchitoperations.techtarget.com/tip/Evaluate-the-benefits-drawbacks-of-service-mesh-technologies (accessed Apr. 29, 2021).

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Masaki Ihara

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Service Mesh Security in Microservices Architecture", supervised by Mohammad Tariq Meeran
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

17.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.