TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Abdulhakim Alsharif - 184070IVSB

Secure Data Exchange Between Microservices

Bachelor's thesis

Supervisor: Muhidul Islam Khan

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Abdulhakim Alsharif - 184070IVSB

Mikroteenustevaheline Turvaline Andmevahetus

bakalaureusetöö

Juhendaja: Muhidul Islam Khan

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: Abdulhakim Alsharif

17.05.2021

Abstract

There are many research papers worldwide that address the wide microservice topic, however, not much research is done in terms of microservice security, or more particularly that in terms of service to service communication.

More than that, companies that are in the process of migrating from a monolith system into microservices often make the mistake of assuming that the security architecture for microservices is the same as a monolith. This thesis serves to outline the cons of such an approach and also showcase an architecture that is more suited for microservices.

The scope of the thesis is for microservice security, more specifically, the security needs of accessing the microservice as a user and as another microservice, with highlighting the possible gaps in the system, along with architectural choices and the benefits and drawbacks of each. This thesis will also analyze companies that use the presented architecture as a practical consideration.

This thesis is written in English and is 31 pages long, including 5 chapters, 6 figures and 2 tables.

Annotatsioon

Kogu maailmas on palju uurimistöid, mis käsitlevad laia mikroteenuste teemat, kuid mikroteenuste turvalisuse osas pole eriti palju uuritud, täpsemalt teenustevahelise side teenuse osas.

Veelgi enam, monoliitsüsteemist mikroteenustesse migreeruvad ettevõtted teevad sageli vea, eeldades, et mikroteenuste turvaarhitektuur on sama mis monoliidil. Selle lõputöö eesmärk on välja tuua sellise lähenemisviisi miinused ja tutvustada ka mikroteenuste jaoks sobivamat arhitektuuri.

Lõputöö eesmärk on mikroteenuste turvalisus, täpsemalt turvatarve vajadused juurdepääsuks mikroteenusele kasutajana ja teise mikroteenusena, tuues välja süsteemi võimalikud lüngad koos arhitektuuriliste valikute ning igaühe eeliste ja puudustega.

See lõputöö on kirjutatud inglise keeles ja on 31 lehekülge pikk, sealhulgas 5 peatükki, 6 joonist ja 2 tabelit.

List of abbreviations and terms

API	Application Program Interface
AWS	Amazon Web Services
CDC	Change Data Capture
CI	Command Line Interface
CLI	Continuous Integration
CD	Continuous Delivery
DAC	Discretionary Access Control
HTTP	HyperText Transport Protocol
OS	Operating System
TLS	Transport Layer Security
JSON	JavaScript Object Notation
JWT	JSON Web Token
JWA	JSON Web Algorithm
JWE	JSON Web Encryption
JWS	JSON Web Signature
RFC	Request for Comments
OPA	Open Policy Agent
PDP	Policy Decision Point
PEP	Policy Enforcement Point
SSL	Secure Socket Layer

Table of contents

1 Introduction 10

2 Theory Background 11

2.1 Definition of Microservices 11

2.2 Foundations of Microservices 12

2.3 The Promise and Limitation of Microservices 15

2.4 Patterns of Microservices 18

2.5 Security 22

3 Methodology 29

4 Analysis 30

4.1 Zero Trust 30

4.2 Enforcing ACL 31

4.3 Enforcing AC Policies to neighbouring microservices 34

4.4 Practical case studies 36

5 Conclusion 38

References 39

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis 42

List of figures

Figure 1: API Gateway

Figure 2: Sidecar Container

Figure 3: Service Mesh

Figure 4: The Perimeter model

Figure 5: Reusable JWT token

Figure 6: New JWT tokens with STS

List of tables

Table 1: Service Mesh Comparison
Table 2: Mutual TLS vs JWS

1 Introduction

1.1 Description of the Problem

The objective of the thesis is to shed light on the security challenges different microservice-based architectures pose, particularly that relating to microservice to microservice authentication and authorization within internal networks.
There are many research papers worldwide that address the wide microservice topic, however, not much research is done in terms of microservice security, or more particularly that in terms of service to service communication.

More than that, companies that are in the process of migrating from a monolith system into microservices often make the mistake of assuming that the security architecture for microservices is the same as a monolith. This thesis serves to outline the cons of such an approach and also showcase an architecture that is more suited for microservices.

1.2 Outline

The thesis will start off with Chapter 2.
Chapter 2 will go through what we need in order to move on to the analysis, the chapter aims to provide a general knowledge on the vast microservices realm, from the sides of security and also what is needed for the analysis.
Chapter 3 will then detail the methodology of the thesis to achieve the novelty
Chapter 4 will go through the necessary information to address the issue and then showcase real life cases.

2 Theory Background

This section serves to detail the concept of microservices - the microservice architecture, through its underlying principles of virtualization, separation of concerns, and continuous delivery. We also detail the pros and cons of the microservice system.

2.1 Definition of microservices

The concept of a “microservice” is a very abstract one, there is no uniform definition of the term microservice, there exists a variety of definitions. These definitions often build around the concept of service, We will explore some of the most common definitions in this thesis.

Definition 1. “A service is a self-contained unit of business functionality that can be accessed remotely and may consist of other underlying services. Communication between services occurs through network calls rather than system calls”[1]

S. Newman outlines microservices as collections of services built to be independent of each other. These services are also created to model around a culture of automation, decentralize the overall system, abstract away core implementation details, isolate failure, and allow independent deployment of services and their monitoring.

The other most cited definition, quoted by Lewis and Fowler. Both view microservices as “an approach to developing a single application as a suite of small services, with each of those services running on its own set of processes, often communicating with simple communication protocols, such as HTTP, through an API resource. These services are built on business capabilities and are independently deployable by a deployment process that is automated, this is also referred to as CD, there isn’t really a centralized management system for microservices, as they are flexible enough to have different tech stacks such as a different programming language or a different data storage technology”.[2]

Definition 2. The concept of microservice architecture is that which focuses on grouping a set of services with a focus on separation of concerns, devops, and virtualization. This is the definition that we will assume In this thesis.

2.2 Foundations of Microservices

There are several foundations in which microservices build their architecture upon. These technologies and principles allow the existence of microservice architectures; microservices can not come into existence with the absence of these foundations.

Separation of Concerns

For microservices, an often effective principle to approach complex issues is to decompose the issue into smaller, more manageable parts, this is a basic principle found in the Software Engineering realm [3,1].

Information hiding, more often defined as Encapsulation, is the main methodology to achieve separating concerns, it works in such a way that elements that logically relate to each other are grouped together in the system. The common implementation of these services are then refactored to use an interface.

Oftentimes, a program that is built strictly after the separating concerns is referred to as modular, software modularity is a concept that goes back all the way to 1969 [4]. An example of a modular system would be the Unix OS.

Separation of concerns (and modularity in general) has many benefits, some are: decreased complexity, cleaner codebase, simple maintenance procedure. Each individual module can be developed in parallel and modified independently because a change in one component should have no minimal effect on others. Moreover, the implementation details of other modules do not need to be uncovered.

The purpose of separating concerns (or modularity) are:

- improve reliability
- Limit failure propagation
- Easier scale

DevOps

In a recent survey conducted by Leppanen et al. [5] about deployment practices in the industry, it is shown that at the time to market could be as low as 20 minutes (the speed in which a development team can apply a change to production using the standard development lifecycle). The survey proves that the time to market is much faster compared to the legacy practice of infrequent deployments during the year.

DevOps is a recent software engineering discipline that highlights the importance of collaboration between development and operations landscapes in terms of collaboration, the aim is to shorten the development cycles and increase the frequency of software releases. DevOps focuses on the organizational side of the problem, while continuous integration (CI) and continuous delivery (CD) are concerned with the technical aspects of automation.

CI is the process of automating the software building and testing processes whenever changes are detected in a version control repository. CD enhances this process even further by automatically deploying any new successful build to a production environment. The automation of infrastructure is an important component of CD that allows applying the same configuration to any number of nodes. Chef is an example of an infrastructure automation tool. Full transition to CD and full automation is not an easy task.

Cloud and Containerization

The cost of computer hardware, such as CPU, memory, and storage, has been steadily decreasing, while the performance has been improving. Together with the availability of high-throughput networks , these factors made the idea of utility computing (a type of on-demand computing) a possibility.

Platform virtualization is a technology that allows one physical server to run multiple virtual machines, i.e, emulations of a computer system. The software that creates a virtual machine on the actual hardware and abstracts the machine’s resources is called a hypervisor. The main benefits of virtualization are improved hardware-resource utilization, live migration, snapshots, and failover. The performance overhead limits the use of virtualization in certain cases, such as time-critical applications with constant loads.

Virtualization enables cloud computing. Cloud Computing is a paradigm that allows different parties to access a shared pool of automatically provisioned and usually virtualized system resources. The term was popularized in 2006 with the launch of Amazon Elastic Compute Cloud (EC2). The three standard models of cloud computing (in order of increasing abstraction) are infrastructure as a service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The primary deployment methods are a private, public, and hybrid cloud.

Containerization is another trending technology. Containerization is a way to encapsulate an application and its dependencies into a self-contained portable unit called a container. While a virtual machine emulates a whole environment with its operating system, the containers all use the same operating system that the host uses, and therefore, are more lightweight, containerization can be viewed as virtualization on the level of the operating system. Both virtual machines and containers share the same goal of portability: the software delivered in a virtual machine or a container will execute identically on any supported platform without additional configuration or installation effort.

Although containers are an old concept dating back to 1998 (the jaul utility in FreeBSD Linux Distribution), the release of Docker, an open source containerization tool, in 2013 made them mainstream. Containers and microservices fit well together. Docker containers are the default choice for microservice deployment nowadays [6]

2.3 The Promise and Limitation of microservices

Advantages

The main benefits of microservices can be consolidated into the following categories:

Scalability. There are three main techniques for scaling [1,2]: hiding communication latencies (i.e using asynchronous communication and data caching), distribution, and replication. Microservices utilize distribution and replication fully. Furthermore, the microservice architecture allows for optimized selective scaling such that only the types of microservices that are in demand at the moment are scaled up whereas the underused ones can be scaled down. The fact that system load is non-uniform is at the heart of microservice architectures.

In a modular architecture, different modules are most likely to have different system resource requirements: while some modules are very sensitive to CPU, others can be sensitive to memory changes. Deploying such modules collectively requires a more powerful hardware than if deployed separately in specialized instances in a cloud environment. The perk of the Microservice architectures is that they facilitate that later, meaning that Microservices can be scaled independently, which leads to a more economic system of resource utilization.

Development time. Despite the need for a supporting infrastructure, microservices are easy to develop individually. Microservice design improves comprehension by decreasing the complexity of individual services. Microservices can be developed and deployed independently which reduces the overhead of coordination between different developers or teams.

In general, computational resources become cheaper while developers' time does not. This makes the trade-off between time to market and performance optimization more prominent. In the microservice architecture, distributed applications. Microservices emphasize automation to optimize development, deployment, and maintenance efforts.

Technological heterogeneity The technological landscape changes rapidly: concepts, tools, frameworks, and programming languages are abandoned, while new ones appear and gain popularity fast. homogeneity does not scale in an economic manner. For

systems beyond a certain size, even if it is possible to maintain a homogenous system, the price of it increases dramatically.

A microservice architecture promotes technological heterogeneity. Individual Microservices can be implemented in different programming languages and use different frameworks. Various versions of the same microservice can co-exist given that the infrastructure to support that is provided.

Microservice benefits are explicitly taken from the benefits of the underlying principles and technologies such as distributed computing.

Disadvantages

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable" - Leslie Lamport, 1987

All the cons of distributed systems are carried over to microservices. However, there exist solutions that counter these cons.

Fault Tolerance: software fault tolerance is an ability of a computer system to continue to operate acceptably despite the presence of partial failures. Software fault tolerance is a valuable property and a design goal of distributed systems [7]. It is a process where the failure of a component may trigger the failure of some other components, causing a domino effect. Such cases can render the entire system nonfunctional, which goes against the foundations of microservices and distributed systems in general

To prevent such critical failures in the microservice world, a design pattern called circuit breaker [8] is often employed. Similar to how an electrical circuit breaker limits the impact of individual service failures from influencing other services. A circuit breaker can be seen as a wrapper wrapped around a service that adjusts the service network behaviour. If the service fails with too many unsuccessful connections made to a particular service, the circuit breaker activates and temporarily blocks attempts at further connections, which will enforce connection timeouts.

Resilience4j latency and fault tolerance library[9] is a popular implementation of the circuit breaker pattern for the java programming language.

Software testing. Testing large , scalable distributed systems is not an easy task. The repeated cases of long lasting downtime due to failures in the microservice-based systems show the importance of systematic resilience testing[7]. Traditionally, tests are run in a test environment before releasing software into production. Some organisations have started to experiment with this however, one example is Chaos Engineering [10], Chaos Engineering is an emerging discipline concerned with systematic resilience testing of software systems in production environments.

The concept of artificially-injected failures into the production environment and continuously stress-testing the system is at the core of chaos engineering. The scale of such intentional failures can vary in a significant manner. Examples of such failures are the termination of virtual machine instances, latency injection into requests between services, failing requests between services, failing an internal service, and making a big part of the service infrastructure unavailable.

Distributed transactions and data consistency. It can be non-trivial to deploy changes that affect multiple services at a time, such as interface changes and distributed transactions. In the microservice architecture, each microservice owns its data. There Is no monolith database shared between all the services, instead, the data is partitioned into smaller databases that are owned by relevant microservices. A need to update multiple databases that belong to different services is likely to arise, leading to data consistency issues.

ACID properties in a distributed transaction are difficult to achieve. There have been different solutions such as compensation over two-step-commit [7] , this is to allow the application of several distinct changes as a single operation. This also helps in terms of rollback if any of the changes prove to cause issues. Eventual consistency is a consistency model with weaker constraints than continuous consistency. Eventual consistency implies that over time all the replicas converge toward identical copies of each other via updates that are guaranteed to propagate

Infrastructure Complexity. Microservice architecture depends heavily on infrastructure automation.particularly service mesh discovery and management functionalities , the functionality is an essential part for infrastructure. Furthermore, it is essential for a large microservice system to have monitoring tools configured.

2.4 Microservice Patterns

2.4.1 API Gateway

The API Gateway is one of the most used patterns for microservices. It can also be called an ingress API gateway. It usually sits right behind the firewall in the data centre, it acts as the single entry point from outside into the infrastructure where the different microservices reside.

The API Gateway's main duties include request routing and protocol translation.

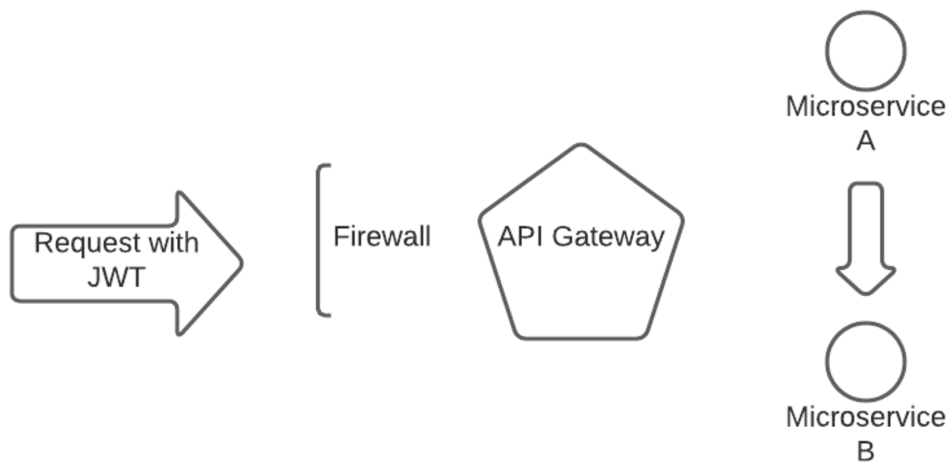

Figure 1: API Gateway Pattern

Request routing and aggregation

The API Gateway acts as a reverse proxy, it routes requests from the clients to the microservices in the infrastructure. When a request is received from the end user, the API gateway forwards it to the designated microservice, this request might include the end user IP address and different request headers, then a response is returned by the microservice to the client.

However, the API Gateway is not like the reverse proxy, the API gateway can also aggregate different results meant for different microservices into a single response. This is called API composition. It address two limitations:

- Provides clients with coarse grained API, therefore it reduces the number of round trips.

- Decouples end users from microservices, however it can couple between the gateway and the microservice

Protocol Translation

In the world of microservices, there exists a large variety of protocols to handle interaction between end user and microservices, and also microservice to microservice. For example , the REST and gRPC protocols are widely used for synchronous requests, while Web Sockets and message based protocols are used for asynchronous requests.

Microservices can also use a wide range of data formats, such as JSON, YAML, protocol buffers or text based formats among others [12]

The API gateway pattern is not perfect, although it solves many issues, it introduces potential issues as well, some may include:

- Potential bottleneck
- Single point of failure
- Additional network latency between end user and microservice
- Must be able to scale up appropriately

Examples of technologies implementing the API gateway include Nginx, HA Proxy, Amazon API Gateway, Traefik.

2.4.2 Sidecar Container

Sidecar containers are a popular pattern that are used to achieve the separation of concerns, discussed in section "Separation of Concerns" topic in section "2.2 Foundations of Microservices", it takes out the common functionality of all microservices into a reusable container that acts a "'sidecar" to the main container that is the application

The sidecar has to be co-located with the maincar, in Kubernetes terms , this would mean as a container with the maincar container in one Kubernetes pod[13]. A pod is the smallest deployment unit in Kubernetes, it encapsulates one or more containers[14], every container in the pod shares network and storage resources and the same lifecycle, meaning that once the main container is destroyed , the sidecar is also destroyed alongside the main container.

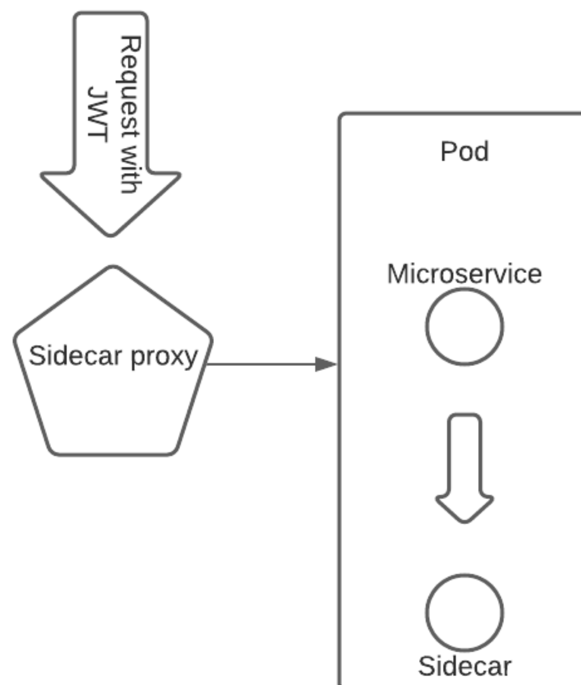

Figure 2: Sidecar Container

An example of the sidecar functionality could be as a service proxy to the main car, it could implement rate limiting, logging, protocol translation, security, and much more.

Running a sidecar container does come with its drawbacks, some may include:

- Higher resource cost
- Containers in the same pod are still slower than language-level method calls[7]

2.4.3 Service Mesh

The service mesh system is a low-latency infrastructure layer that oversees ingress and egress traffic coming to and from each microservice, its job is to secure the traffic with a network security protocol such as TLS, it also handles concerns such as service discovery, load balancing based on rules, and traffic routing [15]

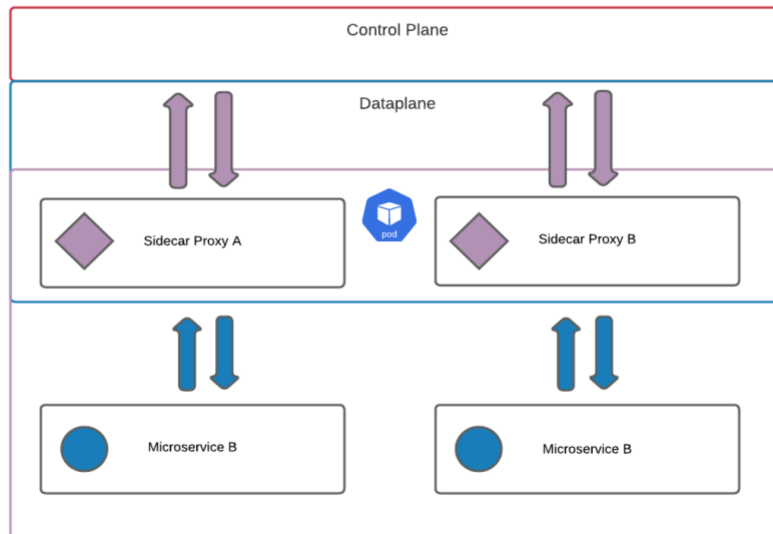

Figure 3: Service mesh system

Service mesh systems are mainly split into two parts, the dataplane and the control plane:

- The data plane contains sidecar proxies which run in each pod alongside the main container, each sidecar proxy provides ingress and egress traffic routing.
- The control plane is a set of APIs that manage and configure the data plane across the service mesh.

In Table 1, we summarise the most widely used service mesh implementation based on the technologies they use.

It is important to note that all of the examples support TLS, HTTP and gRPC protocols along with mutual TLS connections[17,18,19,20]

Service Mesh	Sidecar proxy	Ingress Manager	Containerization Platform
Istio[16]	Envoy	Envoy	Kubernetes
Linkerd[17]	Linkerd-Proxy	Any	Kubernetes
Kuma[18]	Envoy	Any	Kubernetes
Consul Connect[19]	Envoy, Nginx,HAproxy	Envoy,Ambassador	Kubernetes,Nomad

Table 1: Service Mesh Comparison

2.5 Security

Microservice security is not a well studied topic, currently the research dedicated explicitly to microservice security is very scarce. However, the core security principles hold for microservices as well as for any other architecture. A variety of modern security protocols and security best practices can be used when building microservices architecture.

The section starts with explaining the basic security concepts of identification, authentication, authorization, access control, and threat modeling. Next, an overview of the security standards related to public key infrastructure and delegated authorization and authentication is provided. This information is necessary for understanding the solution to the problem presented in the abstract.

2.5.1 Security Concepts

Authentication, Authorization, Identification

Authorization is a process of granting rights to an authenticated entity and specifying what a subject can do. For example, administrative users have more rights available to them than regular users. When a regular user logs in into a system, he/she will not be able to perform administrative tasks such as removing other users from a system. Authorization can also be viewed as the specification of access policies [20]

Identification is a process of claiming an identity of a particular entity without proof, e.g: providing a username. Authentication is a process of confirming the claimed

identity. To be authenticated, an entity provides proof of identity to the authenticating party for verification. The proofs of identity, also known as authentication factors, are based on knowledge, ownership, or inherent properties. Examples of authentication factors are:

- PIN-codes and security questions
- ID cards
- Security tokens
- biometric identifiers

Confidentiality

Confidentiality is a core security that information is not disclosed to unauthorized entities during the information lifecycle. Information may need to be kept confidential in transit and storage, as well as destroyed security. While encryption is used to protect confidentiality of data, other techniques such as MAC or digital signatures are needed to ensure data integrity and authenticity

Threat model

A vulnerability is a weakness in a system that can be exploited by a malicious party. Security threats are usually caused by an exploit of a vulnerability, although other causes such as social engineering and natural disasters are possible. Since it is impossible to be protected from all known and unknown threats, it is necessary to identify possible attack vectors, prioritize the security threats, and plan preemptive actions to avoid the most likely ones. This process is called threat modeling.

An attack vector is a component of a system that an attacker can tamper with, such as input fields and interfaces, to gain further strategic or financial advantage.

An attack surface of a system is the sum of all the existing attack vectors. Attack surface reduction is a known security measure.

2.5.2 Public Key Infrastructure

Symmetric cryptography is the use of the same secret (key) for both encryption and decryption functions. AES is an example of a commonly used modern symmetric algorithm that is fast and secure.

The drawbacks that come with symmetric cryptography mainly come from the difficulty of secure key distribution, large number of keys ($n * (n - 1) / 2$ for a network with n users), and no protection against cheating by the involved parties[21]

Public key cryptography, also referred to as asymmetric cryptography, came to life to address the drawbacks that come with symmetric cryptography. asymmetric cryptography was introduced by W. Diffie, M.Hellman and R. Merkle[22]. Following the same pattern, consider Alice and Bob trying to exchange messages using symmetric cryptography. To encrypt a message for Bob, Alice needs to utilize the public key from Bob, to decrypt the message sent from Alice to Bob, Bob needs his own private key. Both Alice and Bob need to maintain a pair of keys rather than one single key.

The public-key Algorithms are based on the notion of a one-way function such that encryption is computationally easy, but decryption is computationally hard.

The main one-way functions are based either on integer factorization problem (RSA) or discrete logarithm problem (DSA, ECDH)

Symmetric encryption proves to be faster than asymmetric encryption, however, it fails to provide non-repudiation and to provide a secure key. Therefore, the most practical cryptographic protocols are protocols that merge both, they rely on both symmetric and asymmetric algorithms. An example of such protocol would be the SSL/TLS protocols, the cornerstones of secure Internet communication.

Transport Layer Security

TLS , along with its previous iteration, SSL, are a collection of cryptographic protocols that aim to provide communication security, such communication security include preventing eavesdropping, tampering, and message forgery over a computer network. TLS uses a diverse set of algorithms with configurable parameters, mainly for security key exchange, encryption, message authentication, and integrity.

The diversity of available options and presence of legacy components often result into insecure configurations. Open source programming libraries for TLS such as OpenSSL and GnuTLS exist to help programmers implement the protocol. It should be without question that neither TLS nor its different implementations are perfect, RFC7457 lists known attacks against TLS/SSL up to 2015[23]

Digital Certificates and Public Key Infrastructure

Public key encryption requires an authenticated channel for the public keys distribution[21]. The reason why an authenticated channel is needed is to defend against MITM-based attacks, a MITM attack is an attack where an attacker pretends to be a legitimate user by tampering with the communication channel.

Digital Certificates are a viable solution to the problem of public keys authenticity. Certificates attach a public key to an identity by applying digital signatures. Certificates oftentimes are complex in terms of structure and include various fields such as period of validity, issuer, and purpose. The most widely used cryptographic standard is X.509 [24] X.509 is a cryptographic standard that defines the format of public key certificates.

In order to verify a signature of the given message, the receiver of the message must use the public key of the sender. The Certificate Authority (CA) is a mutually trusted third party entity that deals with creating certificates to the communicating parties. CAs can be described as some form of a chain of trust. CAs and supporting mechanisms create a Public-Key Infrastructure (PKI). Running real-world PKI is nontrivial. One of the most challenging tasks of PKIs is certificate revocation. The shortcomings of PKI are discussed in various sources[26] and should be acknowledged.

2.5.3 Delegated Authorization and Authentication

Delegated authorization and shared authentication have become an integral component of modern web security. The most clear and visible mechanism of the concept is social login, which is supported by many web services nowadays.

However, delegated authorization and shared authentication protocols have more practical cases that are interesting, particularly in terms of securing inter-service communication in microservice architecture.

Access Control

Authentication, authorization, and identification are concepts that provide control to all forms of access. There exists many ways of enforcing access control systems, these mainly exist in the ACL model, ACLs are best defined as a collection of permissions that are attached to an object, this object is used to grant access to a specific entity such as a user.

Attribute-Based Access Control

The RBAC model has several pitfalls. In RBAC, the entities are assigned specific roles , these roles are assigned to users based on the functions they perform.

RBAC has some drawbacks, for example, it makes decisions that are based on roles that are statically assigned, therefore RBAC would fall short in a dynamic environment. ABAC further supports the RBAC implementation by having additional attributes, this provides for higher and more granular configuration. It leverages attributes that are associated with the user who made the request [26]. For example, policy rules can be specified in the form of Boolean expressions, Access is granted if the expression is true and denied if otherwise.

Open Policy Agent

OPA is a policy engine that allows ACL policies to be defined and then send them off to be applied to other services. OPA is the central service which decides on policies and their application[27].

OPA supports multiple deployments such as Kubernetes, Istio, Terraform and several of the state of the art technologies.

2.5.4 Delegated Authorization and Authentication

Json Web Tokens, or JWTs are tokens that are capable of containing information about the user that the service can read without calling the issuer. most common ones.

JWT is a RC 7519[28] compliant JSON based object that is used to transmit sensitive information between two parties in a cryptographically safe manner.

STS, or Security Token Service is the entity that handles and forwards claims to be digitally signed. A claim is a piece of information about a subject, usually represented as a name-value pair.

A digitally signed JWT token consists of three base64-encoded parts that are dot-separated:

- Header, this contains the type of the token and the signing algorithm. the type here would be “JWT”, while the algorithm can be specified according to the JSON Web Algorithms (JWA) specification [29]
- Payload. This contains the set of claims. RFC 7519 defines a set of claims, called registered claims, these encode the identity and the metadata, examples may include "iss" , "exp", and "sub".

It is important to note that JWTs are not dependent on any specific technology stack, meaning that they can be used for the entirety of the microservice infrastructure.

2.5.5 Measures

<u>Preventive measures</u>

Addressing the security issue during the early stages of a system is often termed “security by design”, security by design utilises various security principles, some of which are discussed below

- Minimizing attack vector area, meaning, minimizing the number components that are vulnerable to an attacker, such as systems inputs and interfaces, hardens a given system, rendering the exploitation of such a system to be difficult.
- By default, security assumes the fact that the default system configurations and settings are oftentimes left as-is and are rarely modified, primarily due to lack of awareness, convenience, or other possible reasons. Therefore, the focus needs to be on the default configuration. An example would be systems using the least privilege principle by default, meaning, to deny all incoming traffic unless specified otherwise. This would prove to be the safest option.
- Defense in depth implies that no component can be trusted and as much as possible should be verified. Components should not blindly trust each other in a

distributed system. Defense in depth is closely related to another principle called Layering of security mechanisms.

- Least Privilege principle is concerned with limiting the abilities of an entity to the bare minimum required for performing the relevant tasks. The principle expands further to system resource permissions such as network access and CPU and memory allowance.

Detective Measures

In order to fix a security issue, it needs to be detected first. Detection of security accidents is a vital system. System monitoring, including networking monitoring such as firewalls, intrusion Detection (IDS) and Intrusion Prevention (IPS) systems, are examples of detective measures.

- Honeynets are mechanisms to detect and defend against unauthorized access to the system by exposing decoy system components in a carefully controlled environment. Honeynets can distract attackers from actual critical components and assist in worm detection, countermeasures, and spam prevention [30]. honeynets can be used as research tools and to facilitate understanding of the possible attack landscape [30]. Security testing, including penetration testing, can be considered a detective measure.

Corrective Measures

The main objective that corrective measures aim to serve is to minimize the potential damage from a security incident, such as an API token leak, or a critical software bug that has to be fixed as soon as possible. The detection process can be better simplified using logging and monitoring systems. this is mainly for the need for Security Automation and Self-Protection

The required technical knowledge to perform an attack does not follow the same trend as the previous. This is mostly because of automation of the attack process and availability of attack tools.

3 Methodology

The problems that were described in the problem statement are mostly because the whole concept of security in microservices can be one that is riddled with

<u>Outlining the importance of Zero Trust Network policy in the microservice architecture</u>

During Chapter 2, we have discussed the various design patterns we could use for the architecture of microservices. In Chapter 4, we will detail how no matter which pattern is chosen, or which combinations are used, establishing a zero trust network would mitigate low level exploits and achieve greater isolation for microservices.

<u>Detail enforcing ACLs on a general level</u>

This section will aim to give an overview on how ACLs need to be implemented, this section will define the best practices that are about requests that come from the end user to the microservice, this section also shortly explains the two use cases of using new JWT tokens every time, using a STS, or reusing the same JWT token over and over again.

<u>Detail enforcing ACLs on a service to service level</u>

This section will aim to delve into the current solutions to implement the zero trust network policy, specifically taking into account inter service communication. Service to service interaction will be inspected and given recommendations in order to achieve better security.

<u>Analyze how two reputable organisations apply zero trust network policies</u>

As the main goal of the thesis is to showcase the current security trend in microservices, the thesis will outline two companies, Linkedin and Atlassian, that have implemented the zero trust network policy, using ACL and ABAC solutions respectively, utilizing OPA.

4 Analysis

4.1 Zero Trust Security

Traditionally, the network is broken down into zones, this is called the perimeter security model. Each zone is assigned a specific set of permissions. For example, in the edge security architecture similar to the ones.

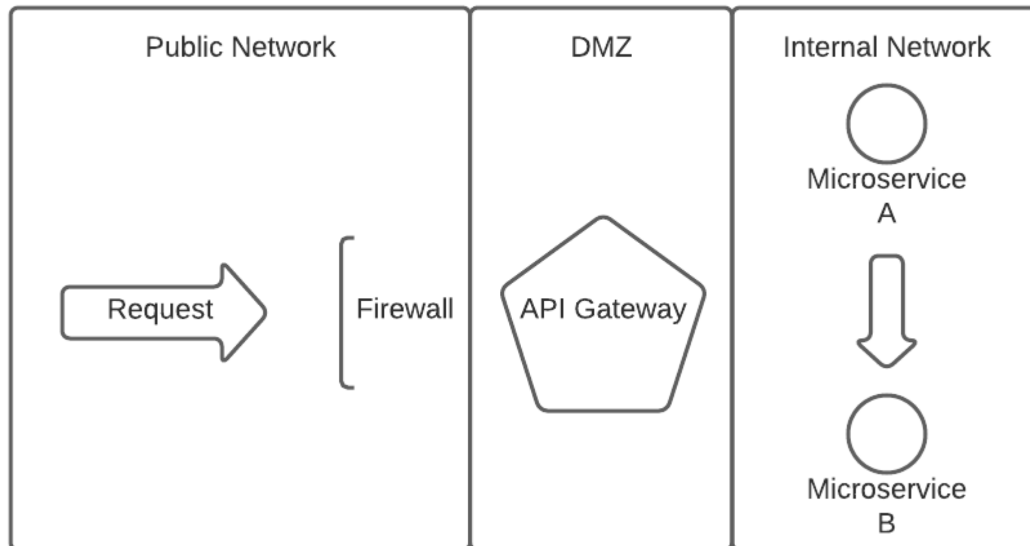

Figure 4: The Perimeter model

The client exists in the untrusted zone, also called the public network, the API gateway in which faces the public internet is placed in the DMZ , which leads to the underlying infrastructure and microservices which are in the trusted zone, as visualized in Figure 4. The perimeter model described relies heavily if solely on network security, that is, placing firewalls between zones with different levels of trust, if left as how it is in a monolith system, it will become highly vulnerable as it lacks intra-zone security[31]

The perimeter model assumes that the private network and each microservice deployed in the private network can be absolutely trusted, this includes everything that a microservice claims to be and the end-user[13]. As we can see this assumption is dangerous and does not address the security threats posed to a microservice in a dynamic environment.

Typically, a microservice-based attack would leverage the perimeter model and involve a microservice in the “trusted” network, using some form of exploit that allows data to be infiltrated from the assumed protected network.

This example is a prime example of the importance of securing microservices that are beyond the edge of the network.

As opposed to the perimeter model, the zero trust model assumes that the network is always hostile. Gilman and barth list five fundamental assertions about a zero trust network[31]:

- The network is never assumed to be trusted
- The network must be secured against both internal and external threats.
- Neither physical nor logical host placement can be used to determine whether the network is trusted or not.
- Every single action, whether it is a user or a microservice must be authenticated and authorized.
- Security policies have to be generated dynamically.

In this context, each request must be authenticated at every entrypoint [13], especially service-to-service interactions, they must be closely monitored continuously in order to confirm the trust given and limit security threats[32].

4.2 Enforcing ACL

All the components of a monolithic system share the same identity, microservices generally do not. In a zero trust microservice system, there needs to be two conditions. First, a microservice must be able to communicate the identity of the end-user to the services that it invokes.

Secondly, a microservice component must not trust a claim made by another microservice about the end-user.

What this means is that the end-user has to be continuously authenticated.

One secure way to allow authentication is to pass the identity of the end-user from one microservice to another in an access token such as JWT.

At this point it's important to note that in the scenario where one JWT is reused (as it shows in Figure)

We could use a new JWT everytime by requesting a new JWT from the STS in exchange for the old JWT, shown by figure 4.2.

The STS must either be the identity service itself or a service that trusts the identity service which issued the JWT. The STS issues a new JWT in exchange for the old JWT under the new microservice that it needs, then it signs it. It can also populate the "jti" claim with a unique identifier for the JWT to prevent the JWT from being replayed [28]. The token exchange approach allows the STS to deny the request if the microservice that called for it is not authorised.

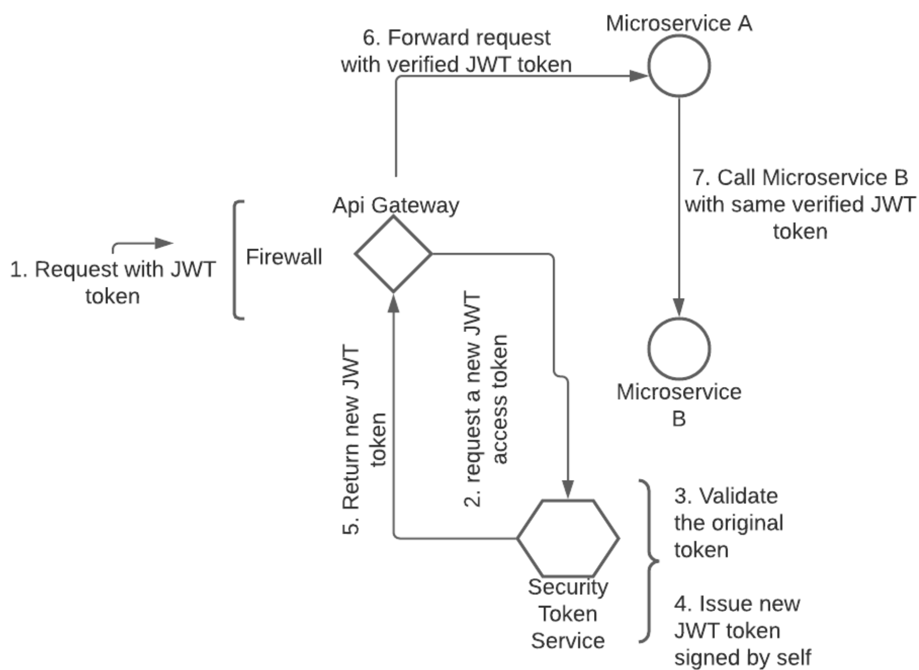

Figure 5: Reusing the same JWT

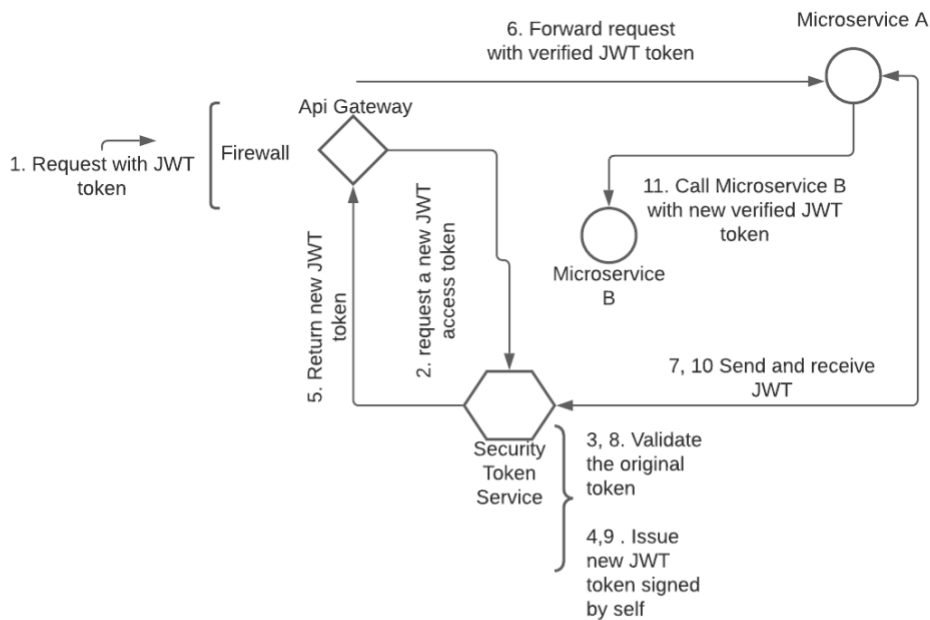

Figure 6: Requesting new JWT

the authorisation implementation could either be handled by the microservice itself like a monolithic application, or it could be sideloaded using a sidecar to adhere to the single responsibility principle.

Delegating authorization to a sidecar has its advantages and disadvantages, those are discussed in the “Sidecar Pattern” found in Chapter 2, neither a firewall or TLS is required between both.

As a best practice for operating containers, the container image has to be immutable, as discussed in “Cloud,Virtualization, and Containerization” topic discussed in Chapter 2 . As a result, authorisation policies need to be updated dynamically . this could be achieved by using a service mesh like Istio[16] , Istio can extract the identity of the end-user from a JWT and allow specific authorisation policies based on that. The authorisation policies are stored in the config store (could be Vault), the policies are then distributed by Istio to each sidecar[16].

The authorisation sidecar can act as the PDP or both the PDP and PEP. In the first case, all microservices can share a library to unify the interaction with the authorisation

sidecar [27]. In the second one, the authorisation sidecar needs to be a sidecar proxy that accepts all the incoming requests before reaching the microservices.

Envoy[33] sidecar proxy is regularly used to authenticate end-users, it supports JWT authentication out of the box [33], while policy engines such as OPA [27] deployed as a sidecar can handle the decision making in terms of authorisation policies.

4.3 Enforcing Access Control Policies on neighbouring microservices

Typically, microservices would interact with one another using HTTP-based REST, WebSockets, GraphQL, and gRPC, etc, regardless of the mechanism, inter-service interaction channels must be protected against snooping and tampering. Microservices must authenticate with one another, complying with zero trust policies[14,32]

Traffic can be secured, inter-service, by using a secure channel such as mutual TLS connections with X.509 Certificates, this is discussed in Section "2.5.2 Public Key Infrastructure". We could also use public-key encryption with signatures such as JWS and JWE.

Mutual TLS's role is to enable microservices to mutually authenticate each other, this protects the integrity and confidentiality of data in transit, but it does not provide non-repudiation. Therefore to enable mutual TLS, we must have valid X.509 certificates, signed by a CA, trusted by all the microservices.

Microservices that are held inside internal networks, usually, a private CA satisfies the security requirement. These are offered by AWS Certificate Manager (ACM), also OpenSSL can be used to set up a private CA as well.

Most service mesh implementations such as Istio and Linkerd support mutual TLS natively[17,18]. Istio for example automatically configures sidecars to use mutual TLS connections when calling microservices [16].

JWS for inter-service authentication

Abbreviated JWS, decodes to contact that is secure with digital signatures. A JWS object contains three concatenated Base64-encoded segments that are dot separated.

JWS does not protect against confidentiality breaches,neither in transit nor at rest. For that we need to use JSON Web Encryption, abbreviated JWE, is the encrypted form of

the JWS signed content, a JWE object contains five concatenated base64 encoded segments that are separated:

- protected header
- encrypted key
- initialisation vector
- ciphertext
- authenticated tag

As shown in Table 2, secure channels offer inter-service authentication, confidentiality, integrity, however they fall short when it comes to non-repudiation compared to public-key cryptographic techniques.

Goal	Mutual TLS	JWS
Inter-Service authentication	True	True
Confidentiality	True	True if JWE is implemented
Integrity	True	True
Non-repudiation	False	True

Table 2 : Mutual TLS vs JWS

- Each microservice is usually handled by a team, this team would also typically manage the permissions for microservice access. This corresponds to Discretionary Access Control (DAC), as the resource is the microservice and the resource owner is the team. This allows complete control over the microservice and what can access it.
- The number of end users far outmatches the number of microservices.

4.4 Practical case studies

LinkedIn[34]

Linkedin is a social network that is aimed towards businesses and professionals,according to the article referenced,as of 2019, Linkedin runs over 700 microservices.

In Linkedin’s case, They have a centralized ACL server, The ACL data is stored in an Espresso database that has a Couchbase cache as its frontend, the data change that is done locally is then delivered using a Change Data Capture (CDC) system, this CDC system listens and captures any changes in the data, then notifies the other services of the change in the original ACL database.

Each service defines a list of services and permissions. There exists an authorisation module , running as a sidecar on each service, this side car keeps a copy of the ACL, this is done in-memory, and then makes an authorisation decision on a local level.

Linkedin also discloses that they use a cloud management interface called Nuage, along with the usual Command Line (CLI) setup.

As the above example used ACL based solution, let’s inspect an approach using ABAC policies for service-to-service authorisation:

Atlassian[35]

Atlassian is a company that mainly specializes in cloud products for software development and project management, this includes Trello, OpsGenie, JIRA, BitBucket, Confluence, and Bamboo.

Atlassian’s products rely on thousands of microservices, with tens of thousands of instances that run in AWS managed regions, on-premise Kubernetes cluster, and others.

Atlassian implements the perimeter security model, as they partition the network into customer, DMZ. and internal zones. Each zone is protected by an edge API or an API gateway, the mentioned APIs enforce security policies on any inter-zone requests.

API Gateways utilize an HTTP request authentication tool named Service Level Authentication (SLAuth). SLAuth also implements authentication on the service level, it uses OPA to achieve authorisation as it does not natively support it.

Policies are written in a JSON-based format, and then translated into Rego, OPA's native object, before policies reach SLAuth, they get submitted to a policy register that validates and tags the policies into an S3 bucket, an S3 bucket is a simple storage service unit provided by AWS. The policies are then searched for via the tags created, all via a CDN.

5 Conclusion

This paper proposed a comprehensive analysis of the issues that pop up during moving from a monolith system to a microservice system, particularly in data exchange between microservices.

In the paper, we have examined how the increased isolation of microservices using a zero trust policy, coupled with design patterns that assist in creating a uniform experience in setting up ACLs and/or ABACs, can lessen the impact of intranet exploitation.

Microservices, when coupled with some method of achieving secure exchange, appear to offer added robustness over monolithic solutions. Key design rules and examples were presented to prove this claim. Furthermore, the analysis of two established companies - Linkedin and Atlassian demonstrated that such a solution is becoming an enterprise standard and that this system when configured correctly, can make the microservice system less vulnerable to low-level attacks.

References

[1]S.Newman "Building Microservices"
https://www.oreilly.com/library/view/building-microservices/9781491950340/

[2]M. Fowler and J. Lewis. "Microservices"
http://www.martinfowler.com/articles/microservices.html

[3]P. Laplante "What Every Engineer Should Know about Software Engineering"
https://www.amazon.com/Every-Engineer-Should-Software-Engineering/dp/0849372283

[4]M. D. Mellroy "Mass produced software components"
http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

[5] Marko Leppanen "The Highways and Country Roads to Continuous Deployment" https://ieeexplore.ieee.org/document/7057604

[6] Jarosław Krochmalski "Developing with Docker"
https://www.oreilly.com/library/view/developing-with-docker/9781786469908/

[7] Brendan Burns "Designing Distributed Systems"
https://www.amazon.com/Designing-Distributed-Systems-Patterns-Paradigms/dp/1491983647

[8]F. Montesi and J. Weber. "Circuit Breakers, Discovery, and API Gateways in Microservices". https://arxiv.org/abs/1609.05830v2

[9] resilience4j https://github.com/resilience4j/resilience4j

[10]IEEE Spectrum "Chaos Engineering Save your Netflix"
https://spectrum.ieee.org/telecom/internet/chaos-engineering-saved-your-netflix

[11] Sam Newman "Building Microservices"
https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/

[12] Protocol Buffers https://developers.google.com/protocol-buffers

[13]P. Siriwardena and N. Dias "Microservice Security In Action" https://www.oreilly.com/library/view/microservices-security-in/9781617295959/#:~:text=Microservices%20Security%20in%20Action%20is,you've%20learned%20into%20production.

[14] Kubernetes Documentation https://kubernetes.io/docs/home/

[15] Chris Richardson "Microservices Patterns" https://www.oreilly.com/library/view/microservices-patterns/9781617294549/

[16] Istio Documentation https://istio.io/latest/docs/

[17] Linkerd https://linkerd.io/2.10/overview/

[18] Kuma Documentation https://kuma.io/

[19] Consul Connect https://www.consul.io/docs/connect

[20] M. Mclarty, R. Wilson, S. Morrison "Securing Microservice APIs" https://www.oreilly.com/library/view/securing-microservice-apis/9781492027140/

[21]C. Paar, J. Pelzel "Understanding Cryptography" https://www.amazon.com/Understanding-Cryptography-Textbook-Students-Practitioners/dp/3642446493

[22] M. Hellman, R. Merkle "New directions in Cryptography" https://ee.stanford.edu/~hellman/publications/24.pdf

[23] Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS https://tools.ietf.org/html/rfc7457

[24] RFC 5280 Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) https://tools.ietf.org/html/rfc5280.html

[25]C. Ellison, B.Schneier https://www.schneier.com/wp-content/uploads/2016/02/paper-pki.pdf

[26]D. F. Ferraiolo "Role-Based Access Control" https://www.amazon.com/Role-Based-Access-Control-Second-Ferraiolo/dp/1596931132

[27] Open Policy Agent https://www.openpolicyagent.org/

[28] RFC 7519 JSON Web Token https://tools.ietf.org/html/rfc7519

[29] RFC 7518 JSON Web Algorithms https://tools.ietf.org/html/rfc7518

[30] P. Gibbs, "Intrusion Detection Evasion Techniques and Case Studies" Sans Institute https://www.sans.org/reading-room/whitepapers/detection/intrusion-detection-evasion-techniques-case-studies-37527

[31] E. Gilman , D. "Barth Zero Trust Networks" https://www.oreilly.com/library/view/zero-trust-networks/9781491962183/

[32] Y. Sun, S. Nanda, and T. Jaeger. "Security-as-a-Service for Microservices-Based Cloud Applications" https://ieeexplore.ieee.org/document/7396137

[33] Envoy Documentation https://www.envoyproxy.io/docs/envoy/latest/

[34]Linkedin at Scale https://engineering.linkedin.com/blog/2019/03/authorization-at-linkedins-scale

[35] Deploying Open Policy Agent at Atlassian

https://www.youtube.com/watch?v=nvRTO8xjmrg&t=1s

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

I , Abdulhakim Alsharif

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Secure Data Exchange Between Microservices", supervised by Muhidul Islam Khan
   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.