Service-to-service Authentication in a Microservice Deployment

Benjamin Ellmer

BACHELORARBEIT

eingereicht am
Fachhochschul-Bachelorstudiengang

Mobile Computing

in Hagenberg

im März 2022

Advisor:

FH-Prof. DI Dr. Marc Kurz

© Copyright 2022 Benjamin Ellmer

This work is published under the conditions of the Creative Commons License *Attribution-NonCommercial-NoDerivatives 4.0 International* (CC BY-NC-ND 4.0)—see https://creativecommons.org/licenses/by-nc-nd/4.0/.

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere. This printed copy is identical to the submitted electronic version.

Hagenberg, March 28, 2022

Benjamin Ellmer

Contents

Declaration **iv**

Abstract **vii**

Kurzfassung **viii**

1 Introduction **1**
1.1 Motivation 1
1.2 Challenges 2
1.3 Chapter Overview 2

2 Microservice Architecture **3**
2.1 Motivation 3
2.2 Comparison to the Monolithic Architecture 4
2.3 Design Principles 4
2.4 Challenges 5
2.5 Usage Situation 5
2.6 Security Consequences 6

3 Related Work **7**
3.1 Microservice Security 7
3.1.1 Edge-level security 7
3.1.2 Service-level security 8
3.2 Public key-Based Authentication 9
3.2.1 Public key cryptography 9
3.2.2 Public Key Infrastructure 10
3.2.3 Key Management 10
3.3 Technologies 11
3.3.1 X509.Certificate 11
3.3.2 JSON Web Token 12
3.3.3 Transport Layer Security 13
3.4 Conclusion 14

4 Authentication Mechanisms **15**
4.1 Authentication based on mTLS 15
4.1.1 Handshake 16

4.1.2 Passing the end user context 17
4.1.3 Conclusion 18
4.2 Authentication based on self-signed JWTs 19
4.2.1 Nonrepudiation 20
4.2.2 Passing the end-user context 20
4.2.3 Conclusion 21

5 Project Structure **22**
5.1 Flea Market App 22
5.2 Communication among the Components 22
5.2.1 Workflow using mTLS 23
5.2.2 Workflow using self-sigend JWTs 24
5.3 Implementation Details 24
5.3.1 mTLS 24
5.3.2 Self-signed JWT 25
5.4 Conclusion 27

6 Experiment **28**
6.1 Setup 28
6.2 Results 29
6.3 Conclusion 30

7 Final Remarks **31**
7.1 Discussion 31
7.2 Summary 32

References **33**
Literature 33
Online sources 34

Abstract

The microservice architecture is a currently emerging pattern in software engineering. Instead of having one huge application, the logic is split into numerous smaller units that fulfill one single purpose. Therefore function calls within the application migrate to remote calls over the network. Remote calls between the services have to provide mutual authentication to secure the system from spoofing. Therefore service-to-service authentication mechanisms are necessary.

The most popular service-to-service authentication mechanisms are self-signed JSON Web Tokens (JWT) and mutual TLS (mTLS). This thesis describes the fundamental concepts and discusses the motivations and challenges of both mechanisms. Furthermore, a project that implements the compared authentication mechanisms is reviewed, and the implementation details are shown. Developers should be able to choose the correct authentication mechanisms for their projects with the knowledge provided in this thesis.

The comparisons showed that both mechanisms are very efficient and provide the same level of security. Therefore none of the mechanisms is superior for all cases. Self-signed JWTs are the preferred authentication mechanism when nonrepudiation is a requirement, when the application tends to share the user-context, or when the developers require to adapt the authentication mechanism with additional parameters. When none of these requirements apply, mTLS is the preferred approach since it keeps the system simple.

Kurzfassung

Die Microservice Architektur is ein aufstrebendes Pattern in der Softwareentwicklung. Eine Applikation welche mit der Microservice Architektur aufgebaut ist, besteht aus vielen kleineren Services, die genau einen Zweck erfüllen. Deswegen werden Funktionsaufrufe innerhalb der Applikation zu Netzwerkaufrufen zwischen den Services. Um die Netzwerkkommunikation vor Spoofing zu schützen muss gegenseitige Authentifizierung gewährleistet werden. Hierfür werden Service-zu-Service Authentifizierungsmechanismen verwendet.

Die verbreitetsten Service-zu-Service Authentifizierungsmechanismen sind self-signed JSON Web Tokens (JWT) und mutual TLS (mTLS). Diese Arbeit beschreibt die Ideen und Konzepte, sowie die Motivationen und Herausforderungen dieser Mechanismen. Außerdem wird ein Projekt vorgestellt, welches beide Mechanismen implementiert und es werden Einblicke über die Implementierung dieses Projekts gegeben. Mit dem Wissen, aus dieser Arbeit sollen Entwickler in der Lage sein selbst den richtige Authentifizierungsmechanismus für ihre Projekte bestimmen zu können.

Die Vergleiche haben gezeigt, dass beide Mechanismen sehr effizient sind und das selbe Maß an Sicherheit bieten. Deswegen ist kein Mechanismus dem anderem gegenüber in jedem Fall überlegen. Self-signed JWTs sind der bevorzugte Authentifizierungsmechanismus, falls Nonrepudiation eine Anforderung ist, wenn die Applikation dazu neigt, den User-Context zu benötigen, oder falls die Entwickler die Authentifizierung selbst mit zusätzlichen Parametern adaptieren möchten. Falls keine dieser Anforderungen zutrifft, ist mTLS der bevorzugte Authentifizierungsmechanismus, da damit das System so einfach wie möglich gehalten wird.

Chapter 1

Introduction

In the past years, a trend towards highly-scalable software systems like the microservice architecture emerged. The migration from a monolithic architecture towards microservices has enormous consequences regarding the security of software systems [15]. Function calls within the same project migrate to remote calls over the network [4]. This offers a larger attack surface because intruders could spoof the communication among the services. Therefore the communication among the services has to provide mutual authentication to prevent attackers from exploiting the system. Additionally, service-to-service communication has to provide confidentiality. Confidentiality is usually addressed using TLS. TLS also provides authentication, but it only authenticates the server to the client. Therefore additional authentication mechanisms are necessary to implement mutual authentication. The most popular approaches for service-to-service authentication are mutual TLS (mTLS) and authentication using self-signed JSON Web Tokens (JWT) [5]. This thesis will describe and compare those authentication mechanisms to discover differences, advantages, and disadvantages.

1.1 Motivation

The International Data Corporation (IDC) has predicted that by 2022, 90% of all apps will feature microservice architectures [23]. So it is inevitable to deal with the numerous mechanisms to secure such systems properly.

Authentication is one of the most crucial security challenges. When authentication is neglected, attackers could perform attacks like the Main-in-the-middle-Attack to exploit the system, even if other security challenges like confidentiality and integrity are mastered. Such attacks could result in substantial data leaks or allow attackers to misuse the system for their advantage.

The microservice architecture is based on having multiple services running in multiple locations. This results in a bigger attack surface because multiple machines are exposed to the internet, making it simpler to find vulnerabilities. This is one of the reasons why Netflix received massive attacks on their microservice based-systems in the past years [14].

These motivations show how vital microservice security is, and this thesis aims to help microservice developers choose the correct authentication mechanisms for their

projects.

1.2 Challenges

Since the microservice architecture became as popular as in the last years, there is a lack of evaluation research and only limited insight into the particular security concerns, especially regarding service-to-service authentication. The most popular solutions and existing implementations are closed source, like the mTLS Architecture of Netflix. Other freely available approaches are often poorly documented and therefore hard to understand [20].

Additionally, the migration from the monolithic architecture to the microservice architecture results in a performance overhead [18]. Furthermore, the authentication mechanisms produce latencies because additional operations have to be performed. Therefore it is crucial to choose the correct authentication mechanism and implement it as efficiently as possible.

The microservice architecture and the service-to-service authentication bring some more challenges and difficulties that will be declared and discussed in the following chapters.

1.3 Chapter Overview

Chapter 2 introduces essential fundamentals of the microservice architecture. It should explain why the later discussed authentication mechanisms are a requirement caused by the microservice architecture.
Chapter 3 summarizes the state-of-the-art concepts regarding microservice security and public key-based authentication. Furthermore, it describes the fundamentals of the required technologies for the later discussed authentication mechanisms.
Chapter 4 describes the fundamentals and concepts of the compared authentication mechanisms in detail. Especially the motivations, challenges, and differences between them are analyzed.
Chapter 5 shows the backend of an app that implements the discussed authentication mechanisms. Additionally, it provides some implementation details for the ASP.Net framework.
Chapter 6 describes the setup of a performance experiment comparing the discussed authentication mechanisms. The experimental results are then interpreted and discussed.

Chapter 2

Microservice Architecture

This chapter introduces the microservice architecture concepts, which are necessary to understand why the later discussed approaches are needed. The principles used to design microservices lead to some characteristics, resulting in the motivations and challenges declared in this chapter. Furthermore, some recommendations about the use cases of the microservice architecture are provided, and the caused security consequences are declared.

2.1 Motivation

Companies like Netflix, Amazon, and Uber are front-runners for building software solutions using the microservice architecture [5]. The main idea is to split the business logic of an application into small autonomous services that work together. This means the programmers have to avoid the temptation of developing too large systems, resulting in the following benefits [12]:

- Technology heterogeneity is achieved through the possibility to use different technology stacks for different services. Depending on the needs of the services, it is even possible to use different data storages for the different microservices (e.g., graph database for users).
- Resilience is achieved since component failures can be isolated, so the rest of the system can carry on working by degrading the functionality of the system.
- Scaling is much more effective due to the possibility to scale only the parts of the system that really need scaling.
- The deployment process is much more convenient because a single microservice can be deployed instead of deploying the whole application, even for small changes.
- The organizational alignment can be improved by assigning the work to small teams that work on smaller codebases, resulting in higher productivity.
- Composability is achieved, considering that the functionalities can be consumed in different ways for different purposes.
- Replaceability is optimized since rewriting a tiny service is much more manageable than replacing a few parts of an vast application.

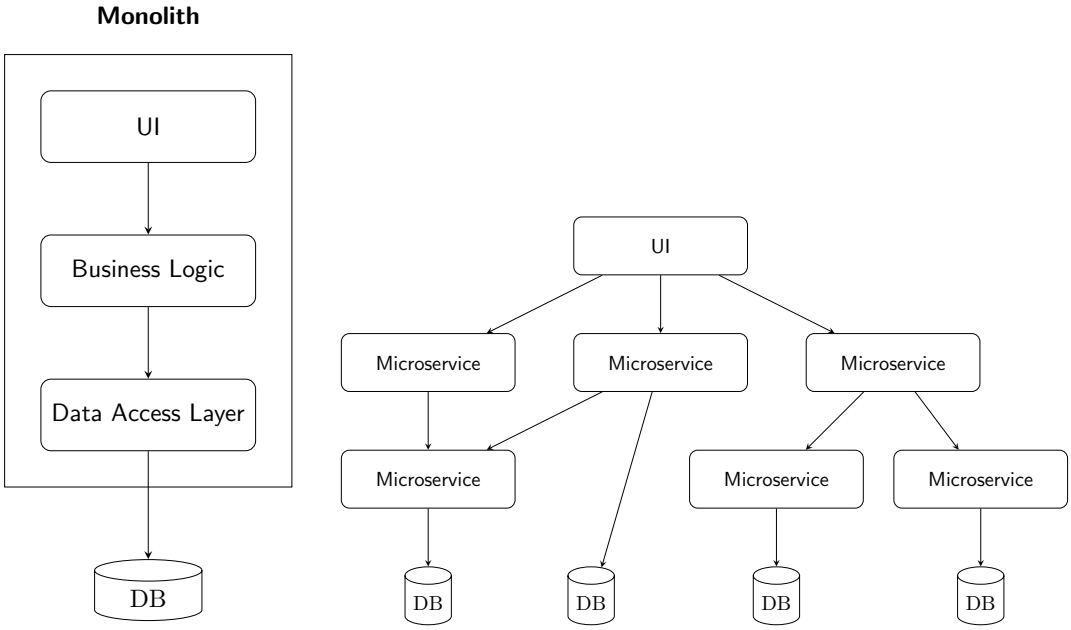

Figure 2.1: Comparison of a monolith application and a microservice based application [9]

2.2 Comparison to the Monolithic Architecture

Figure 2.1 shows the architectural differences between monolithic and microservice applications. A monolithic application has a single unit containing the user interface layer (UI), the business logic layer, and the data access layer. Therefore it is much simpler to manage but brings the following downsides regarding the codebase [9]:

- New features and modifications of old features are harder to implement.
- Refactoring changes can reflect many parts of the software.
- Code duplication raises since it is almost impossible to reuse existing code.

The microservice architecture consists of multiple services focused on only one function of the business logic. Those services communicate with other services using remote calls (e.g., over HTTP), causing additional latencies. Depending on the needs of the services, each service can have its own database that can differ from the database system of the other services. It is also possible to share one database for multiple services, but this should be avoided to reduce coupling.

2.3 Design Principles

It is hard to define principles that will apply to all microservice architectures, but according to Newman, most of them will adhere to the following principles [12]:

Modelled around business concepts: The functionalities are structured around the business contexts instead of the technical concepts.

Adopting a culture of automation: The microservice deployments embrace the culture of automation by using automated tests, continuous delivery, automated servers, and much more automation tools.

Hiding implementation details: The microservices hide as many implementation details as possible to avoid coupling. Especially the databases of the services should be hidden and can be accessed by other services using the APIs of the services.

Decentralising all the things: All approaches that could centralize business logic are avoided to keep associated data and logic within the service boundaries.

Independently deployed: The microservices should provide the possibility to deploy them without having to deploy any other service. Therefore the autonomy of the teams can be increased, and new features can be released faster.

Isolates failures: The microservices have to deal with misbehaving parts of the system and keep on providing as much functionality as possible, to prevent cascading failure.

Highly observable: It is not sufficient to observe a single service's behavior and status. Instead, the functioning of the whole system has to be monitored.

2.4 Challenges

The microservice architecture brings numerous benefits, but it also introduces a set of challenges, that could argue to avoid the microservice architecture in some cases. According to Kalske et al. [9] the microservice architecture causes the following technical challenges:

- The declaration of the service boundaries is very hard [9], especially if the developers do not know the domain very well [12].
- The services should not become too fine-grained to prevent performance overhead. Otherwise, if they are not decomposed enough, one service can affect multiple services.
- Continuous Delivery and Continuous Integration are necessary to manage the services and validate their functionality.
- The integration of the services into other services can become very hard due to the requirement to be available for all used technologies.
- Good logging mechanisms have to be used to recognize failures of microservices as soon as possible.
- Fault tolerance mechanisms have to be implemented to react to situations in which needed services do not respond.

The microservice architecture is gaining popularity, even if it produces so many challenges, showing how crucial its advantages are.

2.5 Usage Situation

According to Newman [12], it is better to start with a monolithic application if the architect does not fully understand the domain and has problems declaring the boundaries

of the services. In such cases, it is better to spend some time learning what the system does first and then break things down to microservices when the system is stable. Furthermore, Newman recommends eliminating legacy monitoring systems before splitting the application into more and more microservices. Otherwise, it could get very messy to stay in knowledge about the system's status.

Fowler [24] recommends considering microservices only when a system gets too complex to manage as a monolith. His essence is to keep the system simple enough to avoid the need for microservices since the microservice architecture can considerably slow down the development. The correlation between the development productivity and the complexity of a system comparing the microservice architecture with a monolithic architecture is shown in figure 2.2.

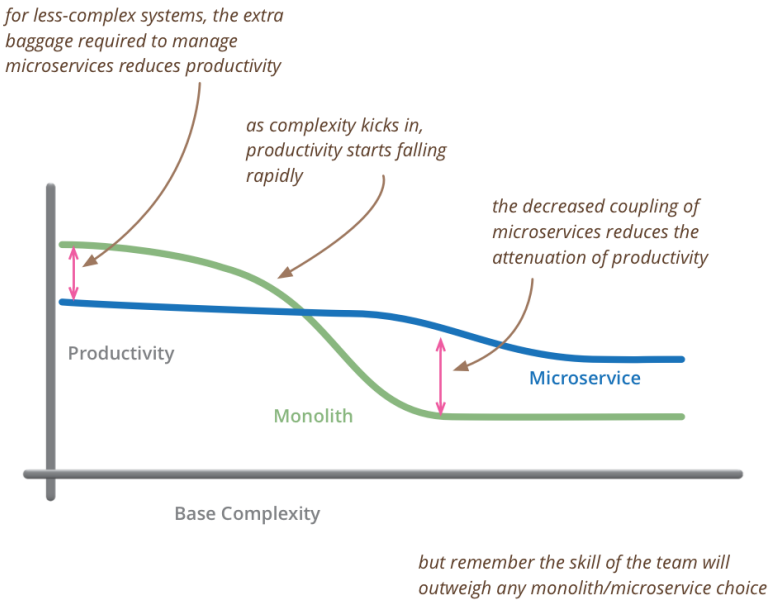

Figure 2.2: Correlation between base complexity and productivity in a microservice architecture compared to a monolithic architecture [24]

2.6 Security Consequences

The migration to the microservice architecture brings many consequences regarding the security of a deployment. Especially the network communications among the services introduce a set of vulnerabilities. Confidentiality, integrity, and availability have to be assured. The need for authentication mechanisms is a common issue of network security, but the migration from language-level calls to remote calls causes the need for mutual authentication between microservices. Therefore, the authentication mechanisms discussed in this thesis are a consequence of the microservice architecture and can be neglected with a monolithic architecture.

Chapter 3

Related Work

This chapter summarizes the state-of-the-art concepts for microservice security, public key-based authentication, and the later required technologies. Service-to-service authentication is only one part of microservice security. Therefore all related topics, considering microservice security, are discussed. Furthermore, the concepts of public-key authentication are explained since the authentication mechanisms compared in this thesis are based on public-key cryptography.

3.1 Microservice Security

Siriwardena and Dias [5] gave an extensive guide of all topics related to microservice security. They separated the security of a microservice deployment into edge-level security and service-level security. Since the microservice architecture splits the backend into multiple minor services, it is insufficient to secure the system only on the edge-level or only on the service-level. Each part of the system must be appropriately secured regarding authentication, authorization, confidentiality, and integrity.

3.1.1 Edge-level security

Edge-level security is defined as the security mechanisms that protect the resources within the deployment from attackers located outside the deployment. The API Gateway is responsible for edge security. Therefore, it is the only entry point to the microservice deployment. It intercepts all requests targeted for the APIs of the services. After validating the requests, it dispatches the valid ones to the microservices. The main tasks of an API Gateway are authentication of the end-user, authorization, and throttling [5]. It authenticates the end-user using access tokens coming from access delegation technologies like OAuth 2.0 or Open ID Connect [16]. By outsourcing the end-user authentication to the API Gateway, the user has to authenticate himself only once and not multiple times to each service [5]. The API Gateway could handle authorization for all requests, but in most cases, it is performed on both levels, the edge-level, and the service level. Therefore the API Gateway is only responsible for coarse authorization assertions, preventing a single-point-of-decision [2].

3.1.2 Service-level security

Service-level security is defined as the security mechanisms that protect the communication among the microservices. According to Barabanov and Makrushin [2], service-level security can be decomposed into the sub-functions service-level authentication, service-level authorization, and external identity propagation. Service-level security can either be implemented by the microservices themselves or by a service-mesh. A service mesh can be seen as a dedicated infrastructure layer that manages the service-to-service communication of containerized services. In a typical microservice deployment with a service mesh, each microservice has its service proxy that works in a transparent manner [5]. The service mesh takes care of service discovery, routing, load balancing, traffic configuration, authentication, authorization, and monitoring [4]. Therefore the services can focus on exactly the tasks they are intended for and do not have to care about security-related tasks [5].

Service-to-service authentication

Authentication is the process of identifying the communication partner to protect a system from spoofing. Since the microservices communicate using remote calls, their communication has to provide mutual authentication [5]. Service-to-service authentication can be implemented in the following ways [5]:

- Trust the Network (TTN)
- Mutual Transport Layer Security (mTLS)
- Self-signed JSON Web Tokens (JWTs)

Trust the Network is a security approach based on the assertion that nobody has access to the components within a network perimeter. All components rely on network security, but internal misbehavior could lead to exploits allowing attackers to intrude into the network perimeter and exploit the microservices [21]. Therefore the industry is heading towards zero-trust networks, and the TTN approach is not more used as primary authentication mechanism [2].

Service-to-service authentication based on mTLS and self-signed JWTs will be discussed in more detail in chapter 4.

Service-level authorization

Authorization defines the tasks that a principal is allowed to perform on a system. It requires that the principal is already authenticated because the authorization is performed based on the identity [16]. Service-level authorization gives the microservices more control to enforce access control. Authorization is usually performed using policy decision point (PDP) models like the centralized PDP model or the embedded PDP model [2, 5]. Proper service-to-service authentication mechanisms are a precondition for service-to-service authorization since the authorization can be bypassed with insufficient authentication [16].

External entity identity propagation

In order to perform the authorization correctly, the services have to know the context of the caller. The most popular technic for identity propagation is extracting the user's context within JSON Web Tokens. The tokens are passed between the microservices and the API Gateway. The propagated identity of the user can be extracted from the token, and the token's signature must be checked. The microservices can then perform authorization based on the identity of the client [2, 5]. The way how identity propagation is implemented depends on the used authentication mechanism. This will be discussed in more detail in chapter 4.

3.2 Public key-Based Authentication

Authentication can be achieved in multiple ways. Two common approaches are symmetric cryptography and public-key cryptography. Both authentication mechanisms discussed in chapter 4 are based on public-key cryptography. Public key cryptography provides higher security than symmetric cryptography. Therefore it is the preferred method to implement authentication mechanisms, although it requires higher computation and communication costs than symmetric cryptography [3].

3.2.1 Public key cryptography

Public key cryptography is also called asymmetric cryptography because the main idea is that different keys are used for encryption and decryption [1]. Each participant is required to own at least one key pair. A key pair consists of a public key available to everyone and a private key that is only known by the owner. Encrypting a message with the public key allows many people to encrypt messages in a way that only the person who owns the private key can read them. Furthermore, it allows one person to encrypt messages in a way that many people can read them by encrypting the message with the private key [7]. This is also known under the term digital signature. Digital signatures can be used to provide authentication and integrity for messages [1].

The commonly used algorithm for digital signatures is RSA. RSA is based on factoring, its encryption key consists of the modulus N, which is hard to factor. The modulus N is calculated by multiplying two large prime numbers p and q with each other. Additionally the encryption key has a public factor e that has no common factors with either $p-1$ or $q-1$. The private key consists of the factors p and q, that have to be kept secret [1]. The person that knows the private key can encrypt a message using the following formula, where M is the message, and C is the encrypted message [1]:

$$C = M^e (modN)$$

An encrypted message can be decryped using the following formular:

$$M = \sqrt[e]{C(modN)}$$

Only the owner of the private key can simply calculate the message from the cipher, using the Fermat's little theorem[1].

[1] See [25] for further information

The problem with asymmetric cryptography is that the encryption and decryption times are worse than using symmetric cryptography. The restriction to use only asymmetric cryptography could decrease the system's performance. Therefore it is common sense to use hybrid systems as done in TLS. In TLS, public-key cryptography is used for authentication, but symmetric cryptography is used to provide confidentiality for the communication [7]. Henriques [7] showed that the performance of systems could be improved using a hybrid approach.

3.2.2 Public Key Infrastructure

Public key cryptography assumes that the receiver of a message already knows and trusts the sender's public key. Public Key Infrastructures (PKI) are used to achieve this assertion. They are responsible for providing a possibility to retrieve the public key in a trusted way. This is done using Certificate Authorities (CA) and certificates. CAs sign certificates, and each communication partner who trusts the CA trusts the certificates signed by it. For this purpose, usually, X.509 certificates are used [1].

A PKI can either be an open PKI (global) or a closed PKI (self-hosted). Closed PKIs have a specific bounded context [8]. A common use case for closed PKIs are microservice deployments because they are usually company intern and have a specific bounded context [5]. The project reviewed in chapter 5 makes use of a closed PKI created with OpenSSH. Closed PKIs are a popular option because they allow risk management and provide secrecy of its code. Open PKIs can be inspected by the public, and based on the inspection, it is determined whether the PKI is trusted or not. Certificates are retrieved by making partnerships with CAs. The main advantage is that no proprietary software is needed since the PKIs are managed by the PKI vendors [8].

3.2.3 Key Management

Key management is a requirement for public key based authentication. It results in being the most challenging part for the later discussed mechanisms. When the key management is fragile, it will have consequences for all parts of the system. Especially service-to-service authentication is affected by the quality of the key management. The authentication mechanism can not stay secure when the keys of the participants are compromised [5, 6]. According to Fumy et al. [6], a key management service has to implement the following tasks:

Entity Registration: The service must provide a procedure to create a link between an authenticated identity and its keys.

Key Generation: The service must provide a procedure to create key pairs with good cryptographic quality.

Certification: The service must provide a procedure for issuing certificates. Certification is often a part of key distribution.

Authentication/Verification: The service must provide a procedure to guarantee entity authentication, message content authentication, and message origin authentication.

Key Distribution: The service must provide a procedure to supply keys for parties legitimately asking for them.

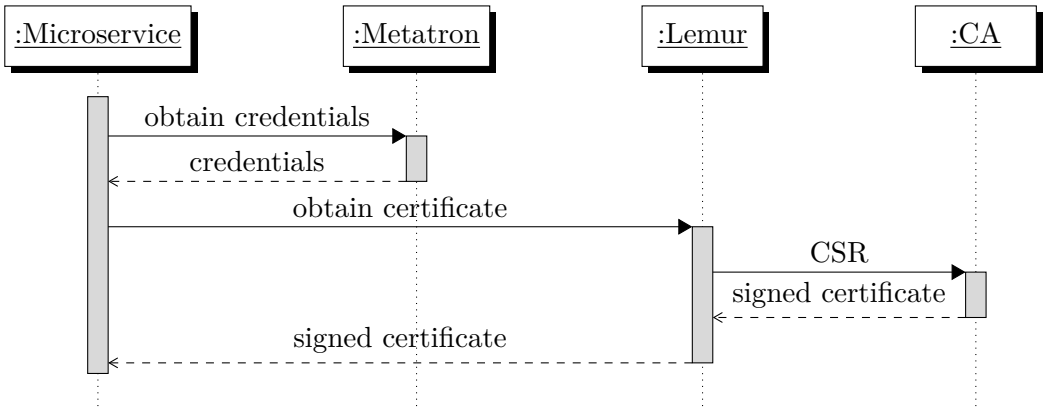

Figure 3.1: Key provisioning of netflix using Lemur and Metatron [5]

Dias and Siriwardena [5] furthermore give insights into the key provisioning (distribution) process of Netflix. Netflix uses its own broker called Lemur for the key provisioning. It is performed in the following steps, visualized in figure 3.1:

1. During the continuous delivery process, each microservice gets a set of credentials that are good enough to access the Lemur APIs. This is done using a Netflix internal tool called Metatron. Metatron credentials are long-lived credentials. They can be used for a longer time period, and the following steps can be repeated multiple times.
2. The microservice talks to the Lemur API to obtain a signed certificate for its credentials. This can happen either during the startup process of the microservice or when the microservice is rotating its keys.
3. Lemur creates a Certificate Signing Request (CSR) addressed to the CA.
4. The certificate is signed using the CA. Lemur is not a CA, but it knows how to integrate with a CA to generate signed certificates.
5. Lemur returns the signed certificate to the microservice, who can then use it to authenticate itself to other services.

Therefore the developers do not have to worry about creating and signing certificates. Instead, they have to implement the communication with the Lemur API.

3.3 Technologies

3.3.1 X509.Certificate

X.509 certifcates bind the subject of a certificate to a public key. They are used to assure the user of a certificate that the certificate's subject owns the corresponding private key. The most significant advantage of certificates is that they can be exchanged using untrusted communication channels because the signature becomes invalid when the content of a certificate is changed. Therefore manipulations can be detected, and manipulated certificates can be declined [26].

When a client wants to consume a service hosted on a server, it must obtain the

server's certificate. If the client does not know the public key of the CA who signed the server's certificate, he has to obtain it. Obtaining the public key often results in chains because the client may have to work his way up until he reaches a CA he trusts. Such chains are also called certification paths [26].

Depending on the version of the standard, a certificate can include more or less information. The information is always stored inside the tbsCertificate, signatureAlgorithm, and signatureValue fields and can be expanded using extensions [26].

tbsCertificate: contains the data of the certificate, including the subject, the issuer, the public key of the subject, the validity period, and additional information [26].

signatureAlogrithm: declares the cryptographic algorithm used to sign the certificate. Algorithms are identified by their unique "OBJECT IDENTIFIER". The most commonly used algorithms are the RSA algorithm and the Digital Signature Algorithm (DSA) [26].

signatureValue: contains the value of the digital signature. It is obtained by signing the content of the tbsCertificate, using the algorithm specified in the signatureAlgorithm field. The signature is used to verify the validity of the information embedded in the tbsCertificate field [26].

Certificate Revocation

Certificate revocation is one of the most significant downsides of certificates. It results in requiring to communicate with a centralized authority for each request [5]. A certificate can be revocated for the following reasons [5]:

- The Private key of the CA is compromised
- The Private key of the subject is compromised
- The holder of the certificate is no longer the identity who requested the certificate
- The CA finds out that the parameters provided in the CSR are invalid

The hardest part about certificate revocation is informing all participants about the revocation of a certificate. This is done using certificate revocation lists (CRL) or other revocation mechanisms. The big downside of CRLs is that each CA has to store a list of revocated certificates and the clients have to retrieve this list whenever they establish a connection to a server. This causes high latencies that can be reduced by caching. Otherwise, caching also reduces security because a certificate can be revocated during the lifetime of the cache. Especially when the CA does not respond to the CRL query is tough to handle [5].

3.3.2 JSON Web Token

A JSON Web Token (JWT) is a container that can carry authentication assertions, authorization assertions, and further information in a cryptographically safe manner. An authentication assertion can be anything that identifies the user. Usually, usernames or e-mail addresses are used as authentication assertions. An authorization assertion can be any information about the access permissions. For example, a JWT can include the information, whether the user is an admin or an unprivileged user [5].

Structure

A JWT is decomposed into the header, the payload, and the signature, which are concatenated and separated by a dot [22].

The **header** contains metadata related to the JWT. The metadata is usually the type of the token and the signature algorithm. The specification defines that only HS256[2] and the none algorithm must be implemented by conforming JWT implementation. It is recommended to additionally implement the algorithms RS256 and ES256[3] [22, 27]. The base64 encoded header is the first part of the JWT.

The **payload** is a set of registered and custom claims. A claim is a piece of information about an entity. The JWT specification defines registered claims that are not mandatory for all cases but should provide a good starting point for a set of valuable claims to ensure interoperability. The software architects can define custom claims on their own, depending on their needs. The custom claims registered in the IANA registry are called public claims, and those not registered in the IANA registry are called private claims [22, 27]. The base64 encoded payload is the second part of the JWT.

The chosen signature algorithm signs the base64 encoded header, the base64 encoded payload, and a secret (only with symmetric encryption like HMAC). The **signature** provides integrity for the message, and if it was signed with a private key, it additionally provides authentication [22]. This mechanisms is later used for service-to-service authentication using self-signed JWTs. The base64 encoded signature is the third part of the JWT.

3.3.3 Transport Layer Security

The Transport Layer Security (TLS) Protocol provides authentication, integrity, and confidentiality for the communication between two parties. It consists of two layers, the handshake protocol and the record protocol [17].

Handshake Protocol

The handshake protocol is responsible for negotiating a cipher suite and providing authentication using X.509 certificates. The cipher suite declares the key exchange algorithm, the signature algorithm, the symmetric encryption algorithm, including the mode of the encryption algorithm and the hashing algorithm [11, 17]. The handshake varies on the key exchange method, but it can be separated into the following steps [10]:

1. The server and the client exchange Hello messages.
2. The server sends its certificate to the client.
3. The client sends a pre-master secret to the server.
4. The client and the server finish the handshake, using the independently computed master secret.

[2] HMAC SHA-256

[3] Elliptic Curve Digital Signature Algorithm (ECDSA) with 256-bit key

Record Protocol

The record protocol provides a secure channel for the communication between two parties. This is done by using the algorithms declared in the cipher suite. Confidentiality is assured using symmetric encryption. Integrity is provided using Message Authentication Codes (MAC) [10, 11].

mTLS

TLS itself is also called one-way TLS because it allows the client to identify the server but not the server to identify the client. Therefore mTLS was introduced to provide authentication in both directions. The server and the client must own a private/public key pair. Therefore it is more appropriate for the communication between systems instead of the communication between users and servers [5]. When mTLS is used, the client must present his certificate to the server during the TLS handshake. The detailed steps are discussed in more detail in chapter 4.

3.4 Conclusion

This chapter described the fundamentals to understand the details and differences between the later described authentication mechanisms. Furthermore, it was described why service-to-service authentication is needed and what additional mechanisms have to be implemented to secure a microservice deployment.

It is essential to understand that the whole system is compromised when one security mechanism does not work properly. For example, authorization can be affected when authentication is neglected. Each service and each layer of the deployment has to be appropriately secured. The problem with the microservice architecture is that one compromised service can be used to compromise other services. Therefore, it is crucial to choose the correct security mechanisms and understand how they work.

Chapter 4

Authentication Mechanisms

This chapter explains the concepts and details of the two compared authentication mechanisms. Only mutual TLS (mTLS) and self-signed JWTs are discussed in more detail since the Trust the Network (TTN) approach is deprecated and should not be used anymore [5]. Additionally, the most popular approaches for user-context sharing are discussed since the chosen authentication mechanisms affect how the user-context is shared.

4.1 Authentication based on mTLS

Mutual TLS is the most popular option for service-to-service authentication in microservice deployments [5]. Securing the communication with TLS already provides integrity confidentiality and authenticates the server to the client. Since TLS does not authenticate the client to the server, it is insufficient for service-to-service authentication. Therefore the efficient and straightforward approach mTLS is used to authenticate the client to the server.

Authentication using mTLS requires a Public Key Infrastructure (PKI 3.2.2), same as TLS on the internet. It is possible to use the already existing PKI of the internet, but this would make the key management much harder and would not bring significant advantages. Therefore it is good practice to use a self-hosted PKI to have a root of trust within the network [5].

When mTLS is used, the server and the client must provide a valid certificate to create a communication channel. The issuer of the presented certificates must be trusted by all communicating parties [5]. If one communication partner does not have a valid certificate, the communication is neglected. Therefore each service needs its private key and the corresponding public key. Additionally, a signed certificate that binds the public key to the certificate's subject is needed. The certificates of the communication partners are exchanged during the TLS handshake.

This mechanism can also be used to authenticate the end-users of an application. The term Client Certificate Authentication (CCA) is used for this context [13]. Service-to-service authentication using mTLS is an implementation of CCA, but in this approach, the client is not the end-user, instead, it is another service.

4.1.1 Handshake

The TLS handshake is used to exchange the certificates of the participants and initialize their connection. The handshake steps differ between the used algorithms and versions of the TLS protocol. The following sequence and figure 4.1 should give an overview about the steps of the TLS handshake using mutual TLS [13]:

1. The client initializes the connection by sending a **ClientHello** message to the server. The **ClientHello** message includes a list of supported cipher suites and the randomness. The randomness is a combination of random bytes and the current date [28].
2. The server chooses one cipher suite of the **ClientHello** message, within the **ServerHello**. Furthermore, the **ServerHello** contains the server's randomness.
3. The server sends the **Certificate** message, containing one or more certificates, that can be used to build the certificate chain. The client validates the sent certificates with his truststore. If it trusts the sent certificate chain, the handshake is continued.
4. The server sends the **CertificateRequest** message, providing a list of all CAs he trusts. The client can use this list to choose the correct certificate he has to present for the Client Certificate Authentication.
5. The server sends the **ServerHelloDone** message, signalizing that the server finished his Hello message.
6. The client responds with his **Certificate** message, which is similar to the servers **Certificate** message, but contains the client's certificate chain.
7. The client generates a random value, the pre-master secret, that is later used to derive symmetric keys for the cryptographic operations defined in the cipher suite. The pre-master secret is encrypted using the server's public key. Therefore, only the owner of the corresponding private key, the server, can decrypt this message. In the end, the encrypted pre-master secret is transferred to the server within the **ClientKeyExchange** message.
8. The client has to prove that he owns the corresponding private key of the certificate he sent. Therefore he has to encrypt the hash of all previous messages with his private key. This encrypted hash is then sent to the server within the **CertificateVerify** message. The server can decrypt the hash and can calculate it on his own to check whether the decrypted hash is correct or not.
9. The client sends a **ChangeCipherSpec** message to signal the server that all following messages will be protected with the protection mechanisms defined in the cipher suite.
10. The last message of the handshake, the **Finished** message contains an encrypted has of all previous messages.
11. Same as step 9, but from the server.
12. Same as step 10, but from the server.

The handshake would have almost the same steps when mTLS is not used. Only the **CertificateRequest** message of the sever and the **Certificate** message and the **CertificateVerify** message of the client are unique for mTLS [13].

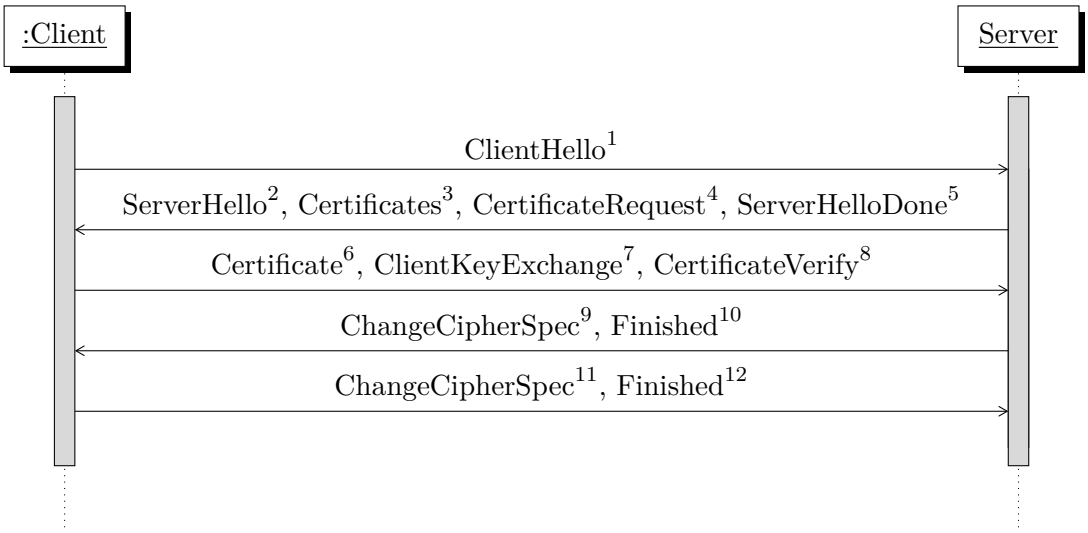

Figure 4.1: TLS handshake using mTLS [13]

4.1.2 Passing the end user context

In some cases, not only the identity of the microservice but also the identity of the end-user is relevant. Then, the microservices have to pass the end-user context when they consume the logic of other microservices. The most popular approach for passing the end-user context are JSON Web Tokens. The JWTs can be embedded within the HTTP Request body or using URL parameters. This approach can be implemented in multiple ways [5].

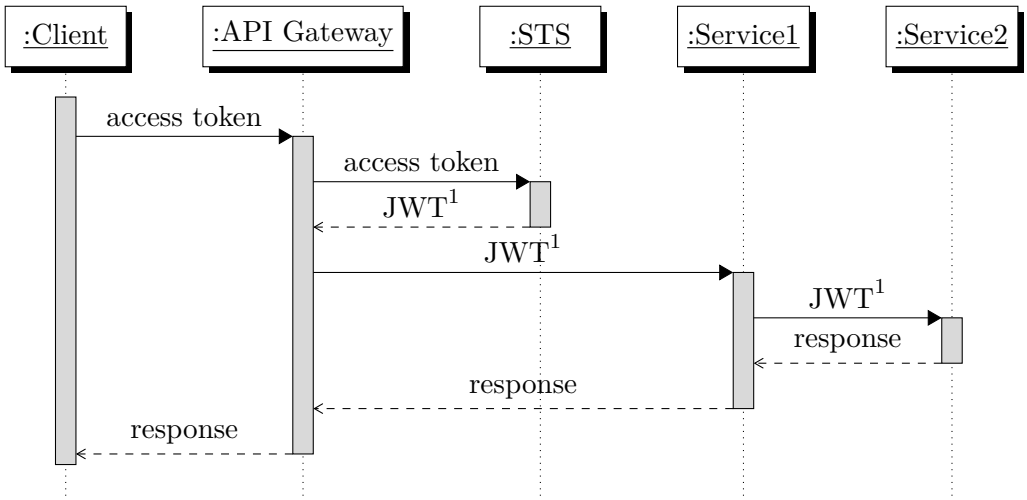

Figure 4.2: Identity propagation using the same JWT for each request [5]

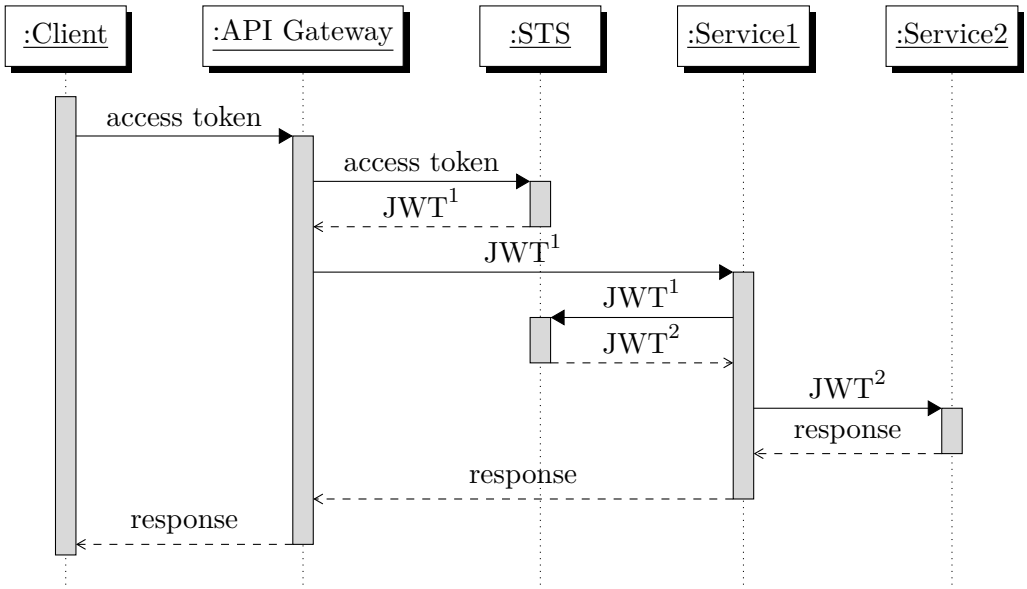

Figure 4.3: Identity propagation using a new JWT for each request [5]

First, the user has to obtain an access token from any token service. This token could be an OAuth2, OpenID Connect, or any other token. Then the user has to embed his token within the Authorization header of each request. The token is validated by a Security Token Service (STS). If the token is valid, the STS returns a JWT that can then be used to consume other services. When one microservice uses the API of another microservice, he embeds the received JWT within the request. If this microservice has to use the API of another microservice, he also passes the same token to the next microservice [5]. The workflow of this approach is shown in figure 4.2

Another approach is that the STS is used to generate a new token for each request, as shown in figure 4.3. When the STS generates a new token for each request, he fully knows all performed requests. Therefore the STS could implement further authorization logic [5]. Nevertheless, the frequent calls could result in an additional workload for the STS and decrease the system's performance.

Both approaches result in some overhead, especially the approach creating a new JWT for each request. Additionally, when the microservices are located in different trust domains, those approaches can get very complicated since a microservice only trusts the STS within its trust domain [5]. The problem with JWTs is that they can be stolen and used by intruders. Therefore it is necessary to provide mutual authentication, confidentiality, and integrity when JWTs are used to share the user context.

4.1.3 Conclusion

mTLS is an efficient mechanism to implement service-to-service authentication. Since the communication between the services is usually secured using TLS, the configuration of mTLS does not cause much overhead, and no new technologies are necessary [5].

A crucial advantage of the TLS handshake is that the private keys are never ex-

changed, and the session keys are always different due to the usage of the randomness. Therefore, even if an intruder can get the session key of a communication channel, he cannot use this key for another session. Furthermore, it is not possible to retrieve any information about the private key of the communication partners with the knowledge of the session key. This shows how secure mTLS is, even for advanced attacks [13].

From the developer's perspective, mTLS does not require much implementation logic. The service that acts as the server has to be configured to use client certificate authentication. Depending on the used technologies, this is usually done by setting a few configuration parameters in the code or directly on the webserver. The service that acts as the client has to be configured to send his certificate during the TLS handshake. Most HTTP Client libraries support simply attaching the certificate to each HTTP request. Nevertheless, the developers do not have to implement much logic resulting in the problem that they do not have much control over the system. The developers have to rely on the implementation of the webserver developers. When a web server has security-related bugs, the microservice developers can not solve them independently. For example, the apache webserver, one of the most popular webservers, had many issues in combination with CCA. Arnis Parsovs [13] researched the problems of the apache webserver and gave an extensive guide on how to circumvent all bugs when CCA is configured.

Key management is the biggest challenge of mTLS, it was described in more detail in chapter 3.2.3. It is responsible for key provisioning, revocation, rotation, and more management tasks. Usually, key management requires a self-hosted PKI for the deployment. For small applications, the key management can be kept very simple. As soon as the deployment grows and many services run simultaneously, automation tools are required. Therefore the management overhead of mTLS is much more complicated to handle than the implementation of mTLS itself [5].

The previously mentioned challenges and motivation result in the conclusion that mTLS is a beneficial and efficient approach when the developers do not require to fully control each aspect of the authentication. Especially when the end-user context has to be shared among the services, mTLS might not be the most efficient solution. mTLS may not be the ultimate tool for all security challenges. Still, when the requirement is service-to-service authentication, mTLS does its job, and it does it well. Therefore mTLS is the most popular approach for service-to-service authentication.

4.2 Authentication based on self-signed JWTs

Self-signed JWTs can be used to provide mutual authentication for service-to-service communication. Same as mTLS, it is based on asymmetric cryptography, and each service needs to own a key pair. The main idea is that the sender creates a JWT signed with his private key. The receiving service can then check the signature of the JWT with the public key of the service that sent the request. The signed JWT is transferred within the Authorization header of the HTTP request [5]. Since the JWT does not have any fixed structure, it is possible to embed contextual data like the user context as claims within the JWT. Therefore parameters and information about the user do not have to be passed within the body or as URL parameters [5].

The usage of self-signed JWTs does not provide confidentiality for the communica-

tion among the services. Therefore TLS should be used to provide confidentiality for service-to-service communication. Since the mechanism is not dependent on TLS, it is possible to use other mechanisms instead of TLS. For example, JSON Web Encryption or any other encryption mechanism can be used, but usually, it does not make sense to get rid of TLS [5]. When TLS is not used, other mechanisms have to authenticate the server to the client.

The usage of self-signed JWTs additionally provides nonrepudiation. Therefore each action is bound to the service that created the JWT, and the service can not repudiate that the JWT was created by him [5]. Whenever nonrepudiation is a requirement of the system, self-signed JWTs are the superior authentication mechanism.

4.2.1 Nonrepudiation

Digital signatures can be used to achieve nonrepudiation cryptographically. Nonrepudiation binds the actions performed by a service to the service or API Gateway who initiated the action. In general, the recipient of a message is provided with proof of the message's origin. Therefore the recipient is protected against situations when a sender denies that he sent a message [19].

Nonrepudability and authentication are very similar. Authentication is about convincing the other party that an event is valid. With nonrepudiation, it is even possible to prove the truth of an event to a third party [19].

A practical example that requires nonrepudiation is an online shop. Assuming that the shop has an **OrderService** and a **PaymentService**. When the payment is accomplished using the functionalities of the **PaymentService**, it has to signalize the **OrderService** that the order was paid. Therefore the **PaymentService** would send a request to the **OrderService** containing a JWT signed using the private key of the **PaymentService**. When the **OrderService** stores the JWT and the request, it can always prove that the request was originated by the **PaymentService**. No other service could have created the JWT because only the **PaymentService** can create a valid signature for his public key.

4.2.2 Passing the end-user context

The mechanisms of how the end-user context is passed between the services are similar to the mechanisms explained in 4.1.2. The big difference is how the JWT is transferred among the services. With mTLS, the JWT was embedded within the body or as a URL parameter, with self-signed JWTs, it can be embedded within the JWT that already has to be transferred. This is done by appending a nested JWT within the claims of the self-signed JWT. The signature of the JWT, which is used for authentication, can still be verified by using the service's public key. The signature of the nested JWT is verified using the public key of the STS. This means the JWT is carried in a way that can not be forged. Therefore it is better to use self-signed JWTs for authentication when the end-user context is relevant in many situations [5].

4.2.3 Conclusion

Service-to-service authentication using self-signed JWTs is not only a mechanism for authentication. It additionally provides nonrepudiation and makes sharing the user context more convenient. Otherwise, the implementation of self-signed JWTs is more challenging than the implementation of mTLS. It is insufficient to configure the webserver correctly and append a certificate to each request. Every service has to know how to encode and decode JWTs. Furthermore, all received JWTs must be stored for an adequate timespan to achieve nonrepudiation, requiring additional database storage.

One significant advantage of self-signed JWTs is that the developers can vary the implementation depending on their needs. For example, the developers can vary the technology used to provide confidentiality. Even if this might not be necessary in most cases, the possibility can be an advantage [5]. Furthermore, the developers can define additional parameters that must be provided within the transferred JWTs. This freedom of choice is not guaranteed with mTLS because it is a strictly defined protocol with very few configuration options.

Sadly the biggest challenge of mTLS, key management, can not be avoided using self-signed JWTs, because it also requires each service to have its key pair. Therefore a PKI is required for both mechanisms. But, the key management can be simplified since it is unnecessary to have a CA responsible for signing each certificate. Still, regarding the webserver setup, it makes sense to have one superior authority, that is the root of trust and whose chained certificates are trusted.

As a result, self-signed JWTs are the preferred authentication mechanism when the target is to achieve nonrepudiation or when the system is very dependent on sharing the user context. Nevertheless, this leads to additional implementation overhead and requires developers specialized in security-related systems.

Chapter 5

Project Structure

This chapter aims to show the concrete structure and communication of a project that implements the previously discussed authentication mechanisms. The focus of this chapter is showing the effects of the different authentication mechanisms on the project itself. For simplicity, this chapter does not handle topics like authorization, key management, and user context sharing. The visualizations are based on the backend of a flea market app.

5.1 Flea Market App

The flea market app is an Android App written in the programming language Kotlin. The main features are buying, renting, and swapping items. The user is authenticated using firebase authentication. Therefore, he must present his access token to the Microsoft API Gateway with each request. The backend of the app is based on the microservice architecture. The microservices are mainly implemented in C# using the ASP.Net Core framework. Actually the backend contains the following services:

- AdService
- UserService
- ChatService
- MediaService
- SubscriptionService
- ReportService

Each service has its own PostgreSQL database. The services are hosted on Microsoft Azure using docker containers. The AdService and the UserService will be used to show the communication within the deployment of the project using the discussed authentication mechanisms.

5.2 Communication among the Components

Figure 5.1 visualizes the communication among the components for an example use case. In this example, the user fetches all ads in his surroundings. Therefore the AdService is used to retrieve the ads from the database, and the UserService is needed to preview

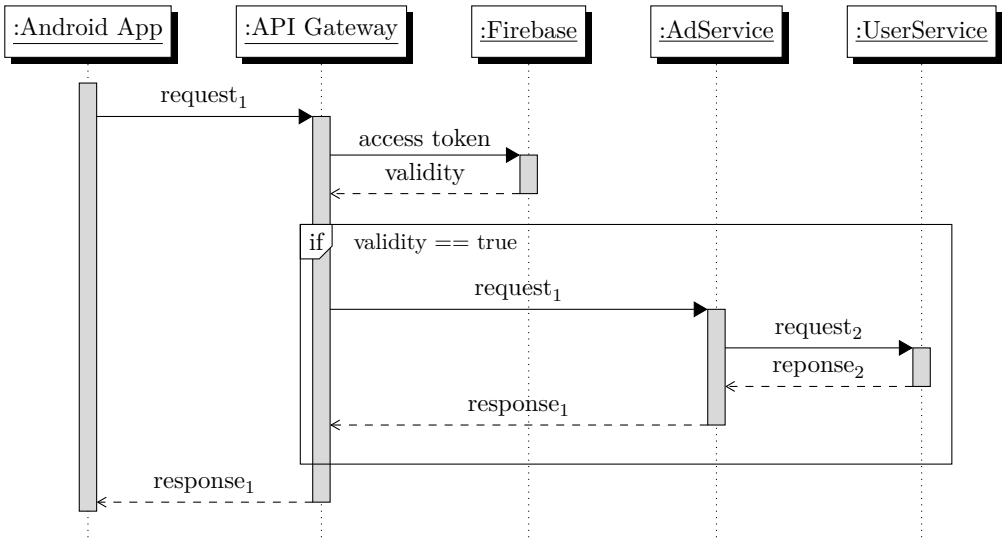

Figure 5.1: Communication among the components of the flea market app

information about the seller. The following components are necessary for the declared use case:

Android App: The Android App is the User Interface for the client that allows the users to access the functionalities of the services. The requests sent by the Android App are sent in beyond of the user.

Firebase: The Firebase service is responsible for validating the access tokens transferred by the users. The app also communicates with the Firebase service to get an access token, but this is done earlier.

API Gateway: The API Gateway is the only entry point into the deployment. Therefore the API Gateway is the only component directly communicating to the Android App.

AdService: The AdService is responsible for managing the users' ads.

UserService: The AdService is responsible for all management tasks regarding the users' data.

5.2.1 Workflow using mTLS

1. The client sends an API request to the Microsoft API Gateway using the Android App. The communication between the API Gateway and the app is secured using HTTPS. Therefore the server is authenticated to the client using TLS. The client is authenticated to the server by embedding a firebase access token within the Authorization header.
2. The API Gateway sends the received token to the Firebase service for validation. This procedure is defined as a policy within the policy file of the API Gateway.

3. The Firebase service returns the information whether the presented token is valid or not.
4. When the client provides a valid access token, the request of the client is forwarded to the **AdService**. The communication between the API Gateway and the **AdService** is secured using mTLS. This means both parties have to present a certificate signed by a trusted CA. Therefore the webserver of the **AdService** has to be configured to allow or even require client certificates.
5. After processing the request from the API Gateway, the **AdService** sends a request to the **UserService** to retrieve additional information about the owner of the ad. The communication between the **UserService** and the **AdService** is also protected using mTLS. Therefore the **AdService** implements the same authentication mechanism as the **UserService**. The certificate is appended to the request using an HTTPClient, that is injected to the Controller of the WebService using Dependency Injection.
6. When the **UserService** processed the request, it responds. The response does not require performing the authentication again since the TCP connection between the **AdService** and the **UserService** is still opened.
7. After the **AdService** processed the response from the **UserService**, it uses the opened connection with the API Gateway and transfers its response.
8. The API Gateway forwards the response from the **AdService** to the Android App. This connection is still secured using TLS. The app can now process the response and present the requested information to the user.

5.2.2 Workflow using self-sigend JWTs

The workflow using JWT and the workflow using mTLS are pretty similar. Therefore, only the steps that differ between the mechanisms are described again.

4. The request from the client is forwarded to the **UserService**. Therefore the API Gateway has to create a valid JWT to communicate with the **UserService**. The JWT, which is signed using the private key of the API Gateway, is transferred within the Authorization header.
5. The **AdService** has to retrieve additional information from the **UserService**. Now the **AdService** has to present his JWT to the **UserService**. This is done by embedding the JWT within the request's Authorization header. If the **AdService** does not know the certificate of the **UserService**, it will deny the request and ask the **AdService** to present its certificate. In this case, the request is repeated, and the certificate of the **AdService** is transferred by appending the certificate as client certificate to the request.

5.3 Implementation Details

5.3.1 mTLS

As already discussed in chapter 4, the implementation of mutual TLS does not require much logic on the service side. The best way to implement mTLS in an ASP.Net

Core API is using the **Microsoft.AspNetCore.Authentication.Certificate** library. It provides the mechanism to add certificate authentication as **AuthenticationScheme** and implement custom **CertificateAuthenticationEvents**. The **CertificateAuthorityService**, implements the interface **ICertificateAuthorityService**, shown in listing 5.1. It provides features to manage a custom certificate truststore and validate certificates using this truststore. An instance of the **CertificateAuthorityService** is injected to the **CertificateAuthenticationEvents**. The dependency injection is managed by the **Microsoft.Extensions.DependencyInjection** library. The **UseAuthentication** and **UseAuthorization** functions of the **ApplicationBuilder** have to be called to use the created **AuthenticationScheme**. The functions of the API Controllers have to be marked with the [**Authorize**] annotation to perform the declared authentication mechanism.

The client has to attach his certificate to the request. In ASP.Net this is done, by creating a custom **HTTPHandler**, and setting the **ClientCertificate** property. The **HTTPHandler** is then used to create a **HTTPClient**, that can be used to perform requests. It is good practice to create the **HTTPClient** once and then inject it into the Controllers using dependency injection. This helps improve the performance and especially prevents the service certificate from being parsed multiple times.

```
1     public interface ICertificateAuthorityService {
2       public void AppendCertificate(X509Certificate2 certificate);
3       public bool ValidateCertificate(X509Certificate2 certificate);
4     }
5
```

Listing 5.1: ICertificateAuthorityService interface that is, implemented by the injected CertificateAuthorityService

5.3.2 Self-signed JWT

The best way to implement mutual authentication using self-signed JWTs in ASP.Net Core is a custom middleware. A custom middleware provides the possibility to perform actions on the receiving request before the API Controllers handle them. The **JWTAuthenticationMiddleware** validates the JWTs and certificates of all received requests. The validation is performed like it is shown in algorithm 5.1. When a client certificate is transferred within the TLS handshake, it is checked using a service that implements the **ICertificateAuthorityService**, shown in listing 5.1. If the certificate is valid, it is mapped to the issuer of the certificate within a custom truststore of the **JWTValidationService**. The **JWTValidationService** implements the **ITokenValidationService** interface shown in listing 5.2. Both the **CertificateAuthorityService** and the **JWTValidationService** are injected into the **Invoke** function of the **JWTAuthenticationMiddleware** using dependecy injection.

eyJhbGciOiJSUzI1NiIsImtpZCI6IkYwMURFNTBCNTU5MzVBQ0VEOUNDNzdCRjRFMjY5NkVDOTE4MUI5NjkiLCJ0eXAiOiJKV1QifQ. eyJuYmYiOjE2NDU1MjU5MjMsImV4cCI6MTY0NTUyNjIyMywiaXNzIjoic2VydmljZTEuc3dhcGluZG8uY29tIiwiYXVkIjoic2VydmljZTIuc3dhcGluZG8uY29tIn0.


```
{
        "alg": "RS256",
        "kid": "F01DE50B55935ACED9CC77BF4E2696EC9181B969",
        "typ": "JWT"
}
{
        "nbf": 1645525923,
        "exp": 1645526223,
        "iss": "service1.swapindo.com",
        "aud": "service2.swapindo.com"
}
```


Figure 5.2: Header and Payload of a JWT, created by the JWTCreatorService

```
1     public interface ITokenValidationService {
2       public void PutIntoTruststore(X509Certificate2 certificate);
3       public bool ContainsIssuerInTruststore(String issuer);
4       public bool ValidateTokenWithTruststore(String token, string issuer);
5     }
6
```

Listing 5.2: ITokenValidationService interface, that is implemented by the injected JWTValidationService

The client has to implement the logic for creating JWTs and attaching them to the requests within the Authorization header. The JWTs are created, using the **JWTCreatorService** that implements the **ITokenCreatorService**, shown in 5.3. The **JWTCreatorService** and a **HTTPClient** are injected to the API Controllers using dependency injection. The Controllers can choose between two clients, one containing the service certificate and one that does not contain a certificate. In both cases, the Controller has to use the **JWTCreatorService** to create a JWT and append it to the Authorization header of each request. The header and the body of an authentication JWT, created by the **JWTCreatorService** is shown in figure 5.2.

```
1     public interface ITokenCreatorService {
2       public string CreateToken(string audience);
3       public string CreateToken(string audience, TimeSpan validityPeriod);
4     }
5
```

Listing 5.3: ITokenCreatorService interface, that is injected to the API Controllers

Algorithm 5.1: Pseudocode of the request validation using self-signed JWTs

```
certificate ← request.Connection.ClientCertificate
token ← request.Headers[Authorization]
missingCertificate ← false
if token is null then
    response.StatusCode ← 403Forbidden
    authenticated ← false
else
    issuer ← null
    if certificate is not null && certificate is trusted then
        putCertificateIntoTruststore(x)
        issuer ← certificate.subject
    else
        if containsIssuerInTruststore(token.issuer) then
            issuer ← token.issuer
        else
            missingCertificate ← true
    if issuer is not null && validateJWT(token) is valid then
        authenticated ← true
    else
        if missingCertificate is true then
            response.StatusCode ← 401Unauthorized
        else
            response.StatusCode ← 403Forbidden
        authenticated ← false
```

5.4 Conclusion

This chapter showed a concrete project that implements the previously discussed authentication mechanisms. Furthermore, some implementation details were shown to give an idea of how the mechanisms can be implemented. The implementation also proved that the authentication mechanism using self-signed JWTs requires more code than the implementation of mTLS.

The concrete implementation of the discussed authentication mechanisms differs depending on the programming language and the used technologies. The main target was clarifying which tasks have to be performed by the services when the compared authentication mechanisms are implemented.

Chapter 6

Experiment

This chapter describes and analyses an experiment measuring the performance of the previously described authentication mechanisms. Even if the performance is not the central decision point to choose the correct authentication mechanism, it is still worth considering. Especially because the migration from a monolithic architecture to the microservice architecture can result in a massive performance decrease of around 79.1% [18]. Therefore the performance is already restricted, and the authentication mechanisms should not produce too much overhead.

6.1 Setup

The setup consists of two components, the service that responds to requests and the client, which performs requests and measures the time taken. Both components are located in the same network and run on the same machine. The service is developed in C# using ASP.Net Core, same as the services of the flea market app, mentioned in chapter 5. The client is a console application developed in C# using .NET. The console application is not a microservice, but it acts as a service located within the deployment. It is not good practice to access the services directly from an external application bypassing the API Gateway. Nevertheless, this makes the experiment setup simpler and reduces falsifications by other components.

The experiment simulates that the client service (the console application) performs 1000 requests to another service. When the first request is performed, the connection between the services has to be established. Therefore the first request will take significantly longer since the full TCP-handshake and TLS-handshake have to be performed. Following requests can reuse the already created connection and do not have to perform the whole initialization process again.

This experiment does not aim to approximate the expected request durations with the declared technology stack. Instead, it will compare the trends between the authentication mechanisms. The exact durations are not relevant since they are influenced by many factors, like the hardware. The tests were performed five times to minimize distortions between the test runs.

Two different implementations of the approach using self-signed JWTs are compared to the performance using mTLS. First, an implementation of the self-signed JWT ap-

proach that creates a new JWT for each request is shown. The second implementation uses the same JWT for all requests. This should illustrate how the performance using self-signed JWTs can be improved by efficiently reusing the tokens. Furthermore, the experiment includes the request duration when only TLS is used. This should show how much time is accounted by the service-to-service authentication mechanisms and how much time is accounted by other tasks like the transport, the random number generation, and the basic TLS-handshake.

6.2 Results

Authentication Mechanism	**Identifier**	**Average Request Duration**	
		first connection	reuse connection
mTLS	α	228.03 ms	9.87 ms
self-signed JWT	β	193.52 ms	13.57 ms
self-signed JWT (reusing token)	γ	184.88 ms	8.10 ms
only TLS	δ	175.78 ms	1.31 ms

Table 6.1: Average request durations of the authentication mechanisms, when random numbers are fetched

The results of the experiment are shown in table 6.1. Furthermore, the trend of the average request duration is visualized in figure 6.1. To simplify the interpretation of the results, the approaches are identified using the identifiers declared in table 6.1.

According to the experiment results, δ is the most efficient approach since it only authenticates the server to the client but not the client to the server. The first request takes only 175.78 ms on average, and the following requests take only 1.31 ms. Comparing δ with α shows that α takes in average 8.56 ms longer than δ. Since the authentication is performed only once per request, it is not valid to say that the request duration between δ and α increases by 750%. The percentual value is only such significant because generating a random number is very simple. Therefore only the time offsets of this comparison are valuable. They show that α increases the request duration by 8.56 ms on average compared to δ. The approaches using self-signed JWTs increase the average request duration by 6.79 ms (γ) to 12.26 ms (β) on average compared to δ.

When mutual authentication is provided, the lowest duration to establish the first connection is achieved using γ. This is reasonable since α has to perform the extended TLS handshake to transfer the certificate and prove that it owns the corresponding private key. When self signed-JWTs are used, the services have to prove that they own the private key of the transferred certificate by signing the JWTs. Therefore the extended handshake is unnecessary for the approaches β and γ. The results show that α takes on average 52.52 ms longer than γ to establish the first connection with a service. γ has a minimal advantage of 8.64 ms on average for establishing the first connection compared to β because it can reuse a previously generated JWT.

Considering the requests after the connection was established, γ is the most efficient approach that provides mutual authentication. Each request takes about 8.10 ms on

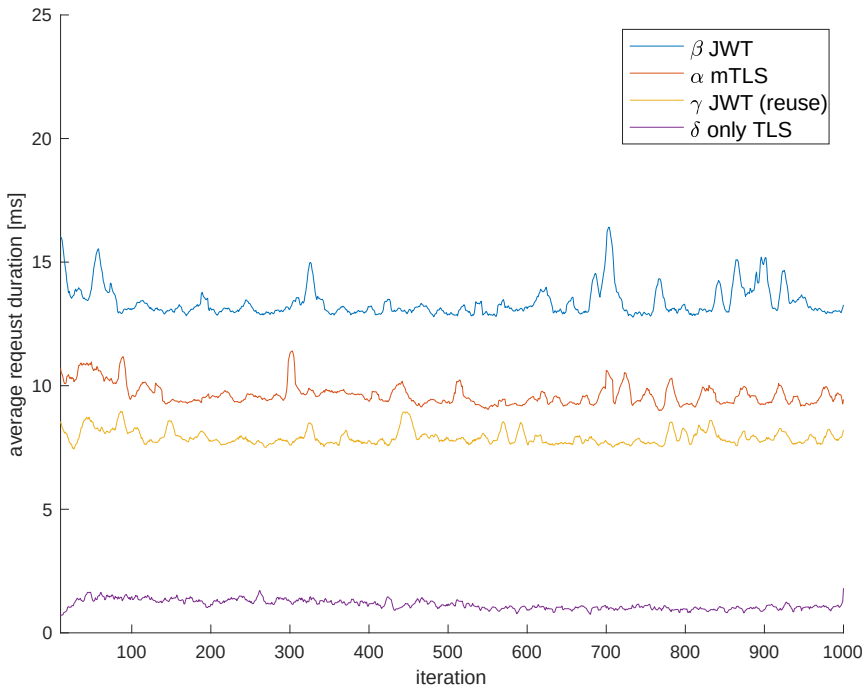

Figure 6.1: Smoothened trend of the average request duration comparing the discussed authentication mechanisms excluding the first request

average, 1.23 ms less than α and 5.47 ms less than β. Nevertheless, it is not possible to reuse the same token for each request in a more realistic example. Therefore using self-signed JWTs would result in a request duration between β and γ, which is about the request duration of α.

6.3 Conclusion

This experiment compared the request durations of the two previous authentication mechanisms. The results showed that the request durations of the mechanism using self-signed JWTs and mTLS are very similar. Depending on the implementation of the JWT approach, it can consume up to 34% more time or up to 20% less time. Therefore, performance is not the most crucial criterion when the two authentication mechanisms are compared because they result in almost similar request durations.

Nevertheless, for time-critical projects where each millisecond matters, self-signed JWTs are the better choice since the performance can be optimized in multiple ways. For example, the certificates of the services can be distributed even before the first request is performed. Or the validity timespan of the tokens can be increased so the same tokens can be used for a longer timespan.

Chapter 7

Final Remarks

7.1 Discussion

This thesis compared and explained the most popular mechanisms for mutual authentication in a microservice deployment. Both mechanisms are based on public-key cryptography, but each mechanism has its own motivations and challenges.

Mutual TLS is an efficient and straightforward authentication mechanism. The implementation of mTLS is very simple since the TLS protocol defines the processes. Otherwise, mTLS does not provide many configuration parameters, and it does not allow to add custom functionalities, like sharing the user context without additional technologies. Nevertheless, if a developer aims to implement only service-to-service authentication, mTLS is the preferred authentication mechanism.

As soon as nonrepudiation is required, self-signed JWTs are the superior authentication mechanism. Furthermore, JWTs make identity propagation more convenient and allow the developers to customize the authentication mechanism and add additional parameters. On the other hand, the implementation of self-signed JWTs is more challenging and requires each service to know how to work with JWTs. Therefore choosing JWTs over mTLS would be unnecessary overhead when the target is implementing only service-to-service authentication. The decision if the additional control of the approach using self-signed JWTs is worth the overhead has to be evaluated for each project independently.

The experiment of chapter 6 showed that the performance of the compared authentication mechanisms is very similar. According to the experiment results, the performance of the approach using self-signed JWTs is very dependent on the implementation of the mechanisms. In some cases, using self-signed JWTs results in a lower request duration, but in some situations, it results in a higher request duration. Therefore the performance is not a criterion that makes any mechanism superior to the other.

The biggest challenge of both authentication mechanisms is key management. Both mechanisms require a PKI and require handling all associated key management tasks. Therefore, implementing the authentication mechanisms is less challenging than the implementation of a key management system. Nevertheless, the level of security that is provided using public-key cryptography is worth the expenses.

7.2 Summary

Service-to-service authentication is a requirement caused by the migration to the microservice architecture. Function calls within the monolithic backend migrate to remote calls. Remote calls have to assure authentication, confidentiality, and integrity. Confidentiality and integrity can be assured using TLS, but authentication of both parties requires additional mechanisms.

This thesis compared two of the most popular authentication mechanisms for service-to-service authentication. Therefore the fundamentals and concepts of the compared authentication mechanisms were described in detail. Additionally, a project using the discussed mechanisms was reviewed, and the consequences of the different mechanisms were described. In the end, an experiment comparing the performance of the discussed authentication mechanisms was performed, and the results were discussed.

References

Literature

[1] Ross Anderson. *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2020 (cit. on pp. 9, 10).

[2] Alexander Barabanov and Denis Makrushin. "Authentication and authorization in microservice-based systems: survey of architecture patterns". *arXiv preprint arXiv:2009.02114* (2020) (cit. on pp. 7–9).

[3] An Braeken. "Public key versus symmetric key cryptography in client–server authentication protocols". *International Journal of Information Security* 21.1 (2022), pp. 103–114. URL: https://doi.org/10.1007/s10207-021-00543-w (cit. on p. 9).

[4] Ramaswamy Chandramouli. "Microservices-based Application Systems". *NIST Special Publication* 800 (2019), p. 204 (cit. on pp. 1, 8).

[5] Wajjakkara Kankanamge Anthony Nuwan Dias and Prabath Siriwardena. *Microservices Security in Action*. Simon and Schuster, 2020 (cit. on pp. 1, 3, 7–12, 14, 15, 17–21).

[6] Walter Fumy and Peter Landrock. "Principles of key management". *IEEE Journal on selected areas in communications* 11.5 (1993), pp. 785–793 (cit. on p. 10).

[7] Michelle S Henriques and Nagaraj K Vernekar. "Using symmetric and asymmetric cryptography to secure communication between devices in IoT". In: *2017 International Conference on IoT and Application (ICIOT)*. IEEE. 2017, pp. 1–4 (cit. on pp. 9, 10).

[8] P Hlavaty. "The Risk Involved With Open and Closed Public Key Infrastructure". *SANS Institute, InfoSec Reading Room Version* 1 (2003) (cit. on p. 10).

[9] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. "Challenges when moving from monolith to microservice architecture". In: *International Conference on Web Engineering*. Springer. 2017, pp. 32–47 (cit. on pp. 4, 5).

[10] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. "On the security of the TLS protocol: A systematic analysis". In: *Annual Cryptology Conference*. Springer. 2013, pp. 429–448 (cit. on pp. 13, 14).

[11] Aleksandr Kurbatov et al. "Design and implementation of secure communication between microservices" (2021) (cit. on pp. 13, 14).

[12] Sam Newman. *Building microservices*. O'Reilly Media, Inc., 2021 (cit. on pp. 3–5).

[13] Arnis Parsovs. "Practical issues with TLS client certificate authentication". *Cryptology ePrint Archive* (2013) (cit. on pp. 15–17, 19).

[14] Anelis Pereira-Vale et al. "Security mechanisms used in microservices-based systems: A systematic mapping". In: *2019 XLV Latin American Computing Conference (CLEI)*. IEEE. 2019, pp. 01–10 (cit. on p. 1).

[15] Ekaterina Shmeleva et al. "How Microservices are Changing the Security Landscape" (2020) (cit. on p. 1).

[16] Prabath Siriwardena. *Advanced API Security*. Springer, 2014 (cit. on pp. 7, 8).

[17] Sean Turner. "Transport Layer Security". *IEEE Internet Computing* 18.6 (2014), pp. 60–63 (cit. on p. 13).

[18] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. "Workload characterization for microservices". In: *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE. 2016, pp. 1–10 (cit. on pp. 2, 28).

[19] Wei Wu et al. "How to achieve non-repudiation of origin with privacy protection in cloud computing". *Journal of Computer and System Sciences* 79.8 (2013), pp. 1200–1213. URL: https://www.sciencedirect.com/science/article/pii/S0022000013000640 (cit. on p. 20).

[20] Tetiana Yarygina and Anya Helene Bagge. "Overcoming security challenges in microservice architectures". In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE. 2018, pp. 11–20 (cit. on p. 2).

[21] Zirak Zaheer et al. "Eztrust: Network-independent zero-trust perimeterization for microservices". In: *Proceedings of the 2019 ACM Symposium on SDN Research*. 2019, pp. 49–61 (cit. on p. 8).

Online sources

[22] Auth0. *JWT Documentation*. URL: https://jwt.io (visited on 10/27/2021) (cit. on p. 13).

[23] International Data Corporation. *IDC FutureScape: Worldwide IT Industry 2019 Predictions*. Oct. 2018. URL: https://www.idc.com/research/viewtoc.jsp?containerId=US44403818 (visited on 10/22/2021) (cit. on p. 1).

[24] Martin Fowler. *Miroservice Premium*. May 2015. URL: https://martinfowler.com/bliki/MicroservicePremium.html (visited on 11/29/2010) (cit. on p. 6).

[25] Dan Ma. *Fermat's Little Theorem and RSA Algorithm*. URL: https://exploringnumbertheory.wordpress.com/2013/07/08/fermats-little-theorem-and-rsa-algorithm/ (visited on 02/27/2022) (cit. on p. 9).

[26] *RFC2459*. URL: https://www.ietf.org/rfc/rfc2459 (visited on 12/29/2021) (cit. on pp. 11, 12).

[27] *RFC7519*. URL: https://datatracker.ietf.org/doc/html/rfc7519 (visited on 10/27/2021) (cit. on p. 13).

[28] *The Beauty of SSL-Handshake*. URL: https://medium.com/@vraghuvaran123/the-beauty-of-ssl-handshake-4286afa543cf (visited on 01/24/2022) (cit. on p. 16).