Service-to-Service Authentication in a Microservice Deployment

Benjamin Ellmer

BACHELORARBEIT

eingereicht am
Fachhochschul-Bachelorstudiengang

Mobile Computing

in Hagenberg

im Januar 2022

Advisor:

FH-Prof. DI Dr. Marc Kurz

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, January 31, 2022

Benjamin Ellmer

Contents

Abstract

This should be a 1-page (maximum) summary of your work in English.

Kurzfassung

An dieser Stelle steht eine Zusammenfassung der Arbeit, Umfang max. 1 Seite. ...

Chapter 1

Introduction

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 [1]. 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. This approach results in the following benefits [5]:

- 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 storage 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 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 a vast application.

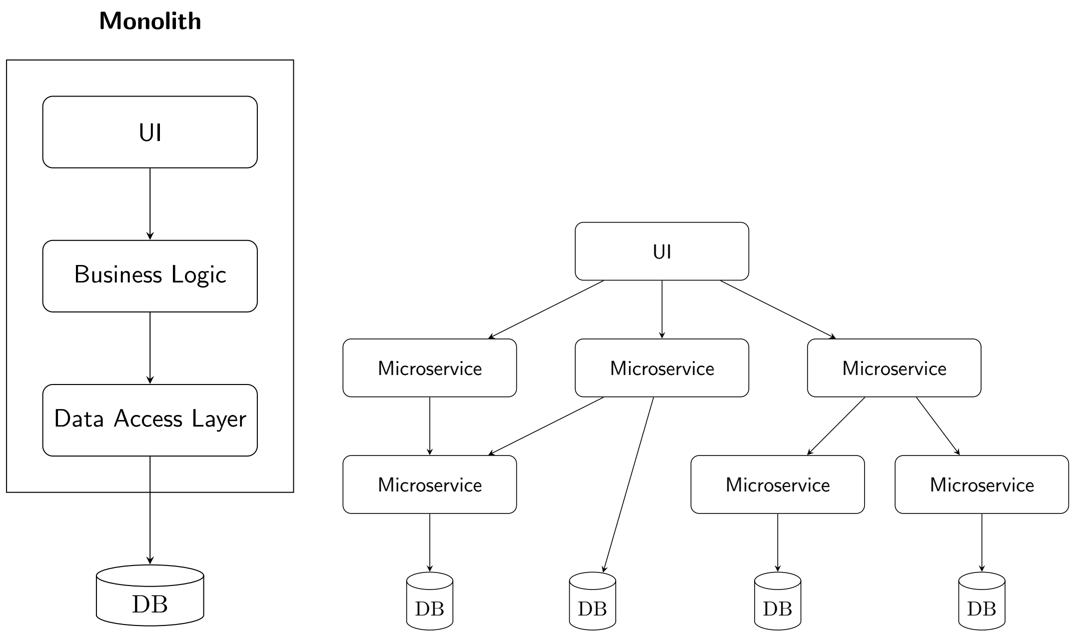

Figure 2.1: Example of monolithic architecture and a microservice architecture [2]

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 [2]:

- 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), which causes higher latencies. Depending on the needs of the services, each service can have its own database, which 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 between the services.

2.3 Design Principles

It is hard to define principles, which will apply to all microservice architectures, but according to Newman, most of them will adhere to the following principles [5]:

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

Adopting 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 APIs.

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

There are some benefits of using the microservice architecture. It also introduces a set of challenges, which could argue to avoid the microservice architecture in some cases. According to Kalske et al. [2] the microservice architecture brings the following technical challenges with it:

- The declaration of the service boundaries is very hard [2], especially if the developers do not know the domain that well [5].
- The services should not become too fine-grained to prevent performance overhead. Otherwise, if they are not decomposed enough, changes to 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 [5] it is better to start with a monolithic application if the architect does not fully understand the domain and has problems declaring the boundaries for 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 [10] recommends considering microservices only when a system gets too complex to manage as a monolith. His essence is keeping the system simple enough to avoid the need of microservices since the microservice architecture can slow down the development considerably. 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.

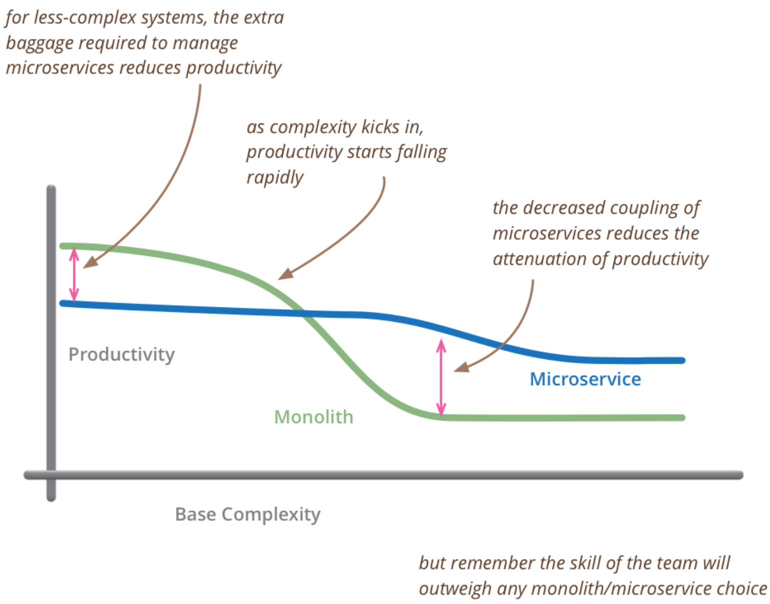

Figure 2.2: Correlation between base complexity and productivity in a microservice architecture compared to a monolithic architecture [10]

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 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

Technologies

This chapter describes the technologies and tools, which are necessary for the implementation of the later discussed authentication mechanisms.

3.1 X509.Certificate

X.509 certificates assure the users of a public key that the associated person or system owns the private key by binding public keys to subjects. Certificate authorities sign certificates and each communication partner who trusts the CA trusts the certificates signed by it. The most significant advantage of certificates is that they can be exchanged using untrusted communication channels because the signatures are not valid anymore when the contents of a certificate are changed. Therefore manipulations can be detected, and manipulated certificates can be declined [11].

3.1.1 Trust Path

When the client of a service wants to consume a service, which is hosted on a server, it has to 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. The way in which the clients can retrieve the CA certificates can be configured by the CA.

3.1.2 Fields

Depending on the version, 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.

TbsCertificate

The TBSCertificate contains the data of the certificate, including the following information:

- Subject of the certificate

- Issuer of the certificate
- public key of the subject
- Validity period
- Additional information

SignatureAlgorithm

The signatureAlogrithm field stores the information, which cryptographic algorithm was used to sign the certificate. Algorithms are declared by their identifier, the "OBJECT IDENTIFIER." The most commonly used algorithms are the RSA[1] algorithm and the Digital Signature Algorithm (DSA) [11].

SignatureValue

The signatureValue field 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.

3.2 JSON Web Token

A JSON Web Token (JWT) is a container, which can carry authentication and authorization assertions and further information in a cryptographically safe manner. An authentication assertion can be anything, which authenticates the user. Usually, usernames or e-mail addresses are used to identify a user uniquely. An authorization assertion can be any information about the access permissions of a user. For example, a JWT can include the information, whether the user is an admin or an unprivileged user [1].

3.2.1 Structure

A JWT is decomposed into the header, the payload, and the signature. The three parts are concatenated and separated by a dot [9]. A valid JWT could look like the JWT shown in figure 3.1.

Header. Payload. Signature

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. eyJzdWIiOiIxMjM0NTY3ODkiLCJpYXQi
OjE1MTYyMzkwMjIsInVzZXJuYW1lIjoiYmVuamFtaW4uZWxsbWVyIiwiZW1haWw
iOiJiZW5qYW1pbi5lbGxtZXJAeWFob28uY29tIiwiYWRtaW4iOmZhbHNlfQ.
0ksqN71
oloNvq3IrY7w72uoTgPz9Gpn08p-KSbFulY0

Figure 3.1: Sample JSON Web Token

[1]Rivest Shamir Adleman

Header

The header contains the metadata related to the JWT, which is usually the type of the token and the signature algorithm. The specification defines that only HS256[2] and none algorithm must be implemented by conforming JWT implementation. It is recommended to additionally implement the algorithms RS256 and ES256[3] [9, 12]. The base64 encoded header is the first part of the JWT.

Payload

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, which are not mandatory for all cases but should provide a good starting point for a set of useful claims to ensure interoperability. Custom claims can be defined by the software architects, 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 [9, 12]. The base64 encoded payload is the second part of the JWT.

Signature

The chosen signature algorithm signs the base64 encoded header, the base64 encoded payload, and a secret. The signature provides integrity for the message, and if it was signed with a private key, it provides authentication [9]. The base64 encoded signature is the third part of the JWT.

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 [7].

3.3.1 mTLS

TLS itself is also called one-way TLS because it helps the client to identify the server, but not the server to identify the client. Therefore mutual TLS (mTLS) was introduced to provide authentication in both directions. The client and the server must own a private/public key pair, so it is more suited for the communication between two systems and not between users and servers [1].

3.3.2 Handshake Protocol

The handshake protocol is responsible for negotiating a cipher suite and for the 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

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

encryption algorithm and the hashing algorithm [4, 7]. The handshake varies on the key exchange method, but it can be separated into the following steps [3]:

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 and if mTLS is used, the client sends his certificate to the server
4. The client and the server finish the handshake, using the independently computed master secret

The steps of the handshake will be explained in more detail in chapter 5.1.1.

3.3.3 Record Protocol

The record protocol provides a secure channel for the communication between the parties. This is done by using the algorithms declared in the cipher suite. Confidentiality is assured, using symmetric encryption, and integrity is provided by Message Authentication Codes (MAC) [3, 4].

Chapter 4

Related Work

Chapter 5

Authentication Mechanisms

This chapter explains the concepts and details of the two compared authentication mechanisms. Only the mTLS approach and the authentication using self-signed JWTs approach are discussed in this chapter since the Trust the Network (TTN) approach is deprecated and should not be used anymore [1].

5.1 Authentication based on mTLS

Mutual TLS is the most popular option for the service-to-service authentication of microservice deployments [1]. Securing the communication with TLS already provides integrity confidentiality and authenticates the server to the client. Since TLS does not provide authentication from the client to the server, it is insufficient for service-to-service security. Therefore mutual TLS is used, which provides an efficient and straightforward approach to authenticating the client to the server.

The authentication using mTLS requires a PKI, the 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 any advantages. Therefore it is good practice to use a self-hosted PKI to have a root of trust within the network [1].

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 [1]. 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, which 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. The 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.

5.1.1 Handshake

The handshake is used to exchange the certificates of the participants and set up the connection. The handshake steps differ between the used algorithms and versions of the

TLS protocol. The following sequence and figure 5.1 should give an overview about the steps of the TLS handshake using mutual TLS [6]:

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, which is a combination of random bytes and the current date [13].
2. The server responds with a **ServerHello**, in which he chooses one cipher suite of the **ClientHello** message. Furthermore, the **ServerHello** contains the server's randomness.
3. The server sends the **Certificate** messages, containing one or more certificates, which can be used to build the certificate chain. The client validates the sent certificates with his own trusted store. If it trusts the sent certificate chain, the server is successfully authenticated.
4. The server sends the **CertificateRequest** message, in which the trusted CAs of the server are listed. The client can use this list to choose the correct certificate he has to present.
5. The server sends the **ServerHelloDone** 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 then generates a random value for the pre-master secret. The pre-master secret derives symmetric keys for the cryptographic operations defined in the cipher suite. Then 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 with the certificate's public key and can calculate the hash on its 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 is the **Finished** message, which is an encrypted hash of all previous messages.
11. Same as step 9, but from the server.
12. Same as step 10, but from the server.

After the handshake, both participants know the secret, which can encrypt and decrypt messages. 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. One special case of the handshake is that the client responds to the **CertificateRequest** with an empty **Certificate** message. Depending on the configuration of the server, the connection without a certificate can be allowed or neglected [6].

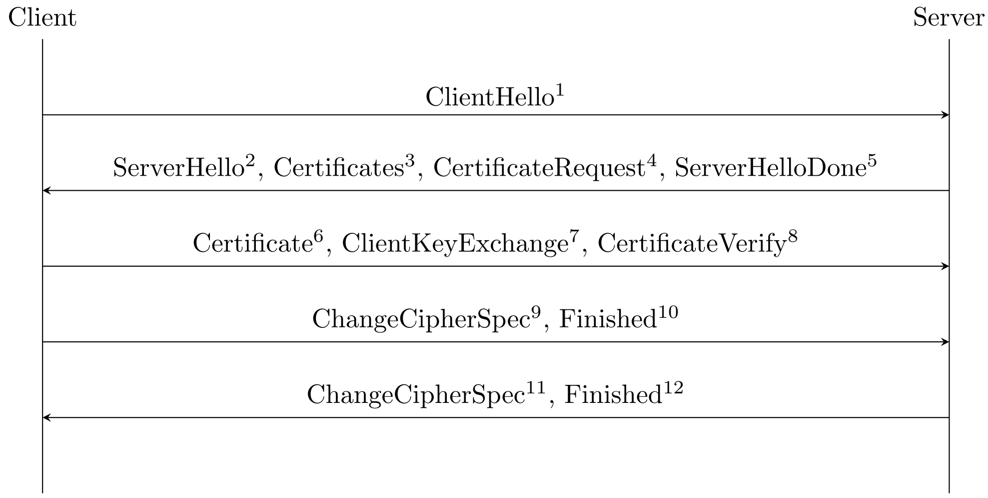

Figure 5.1: TLS handshake using mTLS [6]

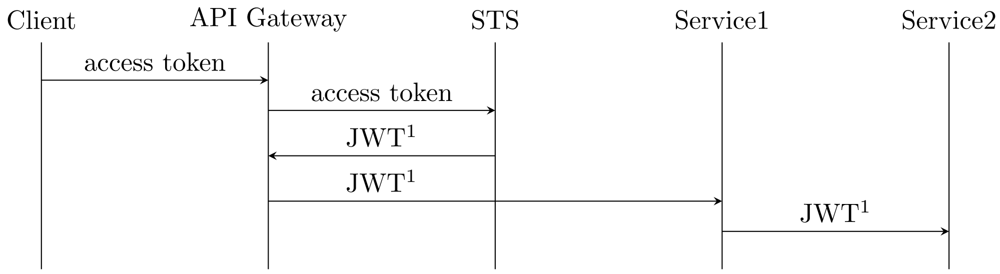

Figure 5.2: Use the same JWT for each request [1]

5.1.2 Passing the end user context

For some functionalities, the identity of the microservice is not relevant. Instead, the identity of the end-user is relevant. In those cases, 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 with mTLS is 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 [1].

Generally, the user obtains an access token from any token service. This token could be an OAuth2, OpenID Connect, or any other token. The user has to send this token with each request. The token is then validated by a Security Token Service (STS). If the token is valid, the STS returns a JWT, which can then be used to consume other services. When one microservice calls another microservice, he sends the JWT, and if this microservice has to consume another microservice, he also passes the same token to the next microservice [1]. The workflow of this approach is shown in figure 5.2

Another approach is that the STS is used to generate a new token for each request which is shown in figure 5.3. When the STS generates a new token for each request, he fully knows all performed requests. Therefore the STS could implement further authorization logic [1]. Nevertheless, the frequent calls could result in an enormous workload for the STS and decrease the system's performance.

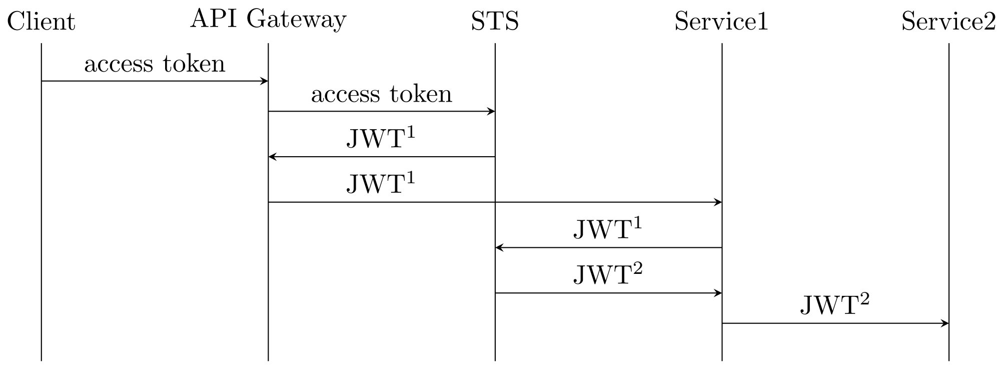

Figure 5.3: Generate new JWT for each request [1]

Both approaches result in some overhead, especially the second one. 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 [1]. Therefore, mTLS might not be the superior mechanism for systems that require the end-user context.

5.1.3 Conclusion

mTLS is an efficient mechanism to implement service-to-service authentication. Since the communication between the services is usually done using HTTPS, the services already use TLS. Therefore, mTLS does not cause much configuration overhead, and no new technologies are necessary [1], unless the services share the end-user context.

One crucial advantage of the TLS handshake is that the private keys are never exchanged, and the session keys are always different due to the usage of the randomness. This means that 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 [6].

From the developer's perspective, mTLS does not require much implementation logic. The service which acts as the server has to be configured to use 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 which 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. This means that if 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 [6] researched the problems of the apache webserver and gave an extensive guide on how to circumvent all bugs when CCA is configured.

The biggest challenge of mTLS is the key management, which was described in more detail in the chapter. Key management is responsible for key provisioning, key revocation, key rotation, and more management tasks. Usually, the key management requires a self-hosted PKI for the deployment. For small applications, the key management can be kept very simple. Nevertheless, as soon as the deployment grows and many services are running simultaneously, automation tools are required. Therefore the management overhead of mTLS is much harder to handle than the implementation of mTLS itself [1].

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. Even if mTLS may not be the ultimate tool for all security challenges regarding service-to-service authentication, it does its job, and it does it well. This is why mTLS is the most popular approach for service-to-service authentication.

5.2 Authentication based on self-signed JWTs

Self-signed JWTs can be used to provide 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, which is signed with his private key. The receiving service can then check the signature of the JWT with the public key of the sending service. The signed JWT is transferred within the Authorization header of the HTTP request [1]. 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 [1].

The usage of self-signed JWTs does not provide any confidentiality for the communication among the services. Therefore TLS should be used to secure service-to-service communication. Since the authentication 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 exchange TLS [1].

The usage of self-signed JWTs additionally provides non-reputability. This means each action is bound to the service that created the JWT, and the service can not repudiate that the JWT was created by him [1]. Whenever non-reputability is a requirement of the system, self-signed JWTs are the superior authentication mechanism.

5.2.1 Non Repudability

Digital signatures can be used to achieve non-reputability cryptographically. This means that when one service acts as another service, the receiving service can prove that the calling service initiated the action. A practical example is that a customer creates an order consuming the “OrderService” and the “OrderService” updates the inventory using the “InventoryService”. The “InventoryService” could save the JWT it received from the

"OrderService". The "InventoryService" can later prove that the update was initiated by the "OrderService" because only the "OrderService" could have created the JWT with a valid signature. Therefore each transaction within the deployment can be comprehended entirely, as long as the services store the received JWTs [1].

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 [8].

5.2.2 Passing the end-user context

The approaches to how the end-user context is passed between the services are similar to the approaches explained in 5.1.2. The big difference is how the JWT is transferred among the services. While 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. Nevertheless, 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 the authentication when the end-user context is relevant in many situations [1].

5.2.3 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 much 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 unless he only sends or receives requests. Furthermore, to achieve nonrepudiation, all received JWTs must be stored for an adequate timespan, requiring additional database storage.

One major advantage is that the authentication using self-signed JWTs can be implemented differently. It does not matter which technology is used to provide confidentiality for the communication among the services. The common way is TLS, and there are not many reasons to get rid of TLS, but the fact that it is possible to get rid of TLS can be an advantage in some situations. It does not matter which technology is used to transfer the public key of the sending service to the receiving service. Moreover, the developers can vary some other parameters, which is not possible using mTLS because it is a strictly defined protocol.

Sadly the biggest challenge of mTLS, which is the key management, can not be avoided using self-signed JWTs, because it also requires each service to have its key pair. Nevertheless, 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 who specialize in security-related systems.

Chapter 6

Project Structure

This chapter shows the structure of an project which implements the previously discussed authentication mechanisms. This chapter aims to clarify the interactions of the components which are needed to implement the service-to-service security. The visualizations of this chapter are based on a microservice backend for a flea market app. The backend is implemented in C# using the ASP.NET framework.

6.1 Components

The components of the example deployment are shown in figure 6.1. It consists of the following parts:

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

Firebase Authentication: The Firebase Authentication service is responsible for validating the access tokens wich are transferred by the users.

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

AdService: The AdService is one service of the deployment. It is responsible to manage all ads offered to the users of the app.

UserService: The AdService is the service which is responsible for managing the data of the users.

6.2 Workflow using mTLS

1. The client sends an API request to the Microsoft API Gateway using an Android App. The client communicates with the API Gateway using HTTPS. Therefore the server is authenticated to the client using TLS. The client is authenticated to the server by embedding an firebase access token within the Authorization header.
2. The API Gateway receives the request from the client and has to check the provided access token. The token is validated using an token validation service pro-

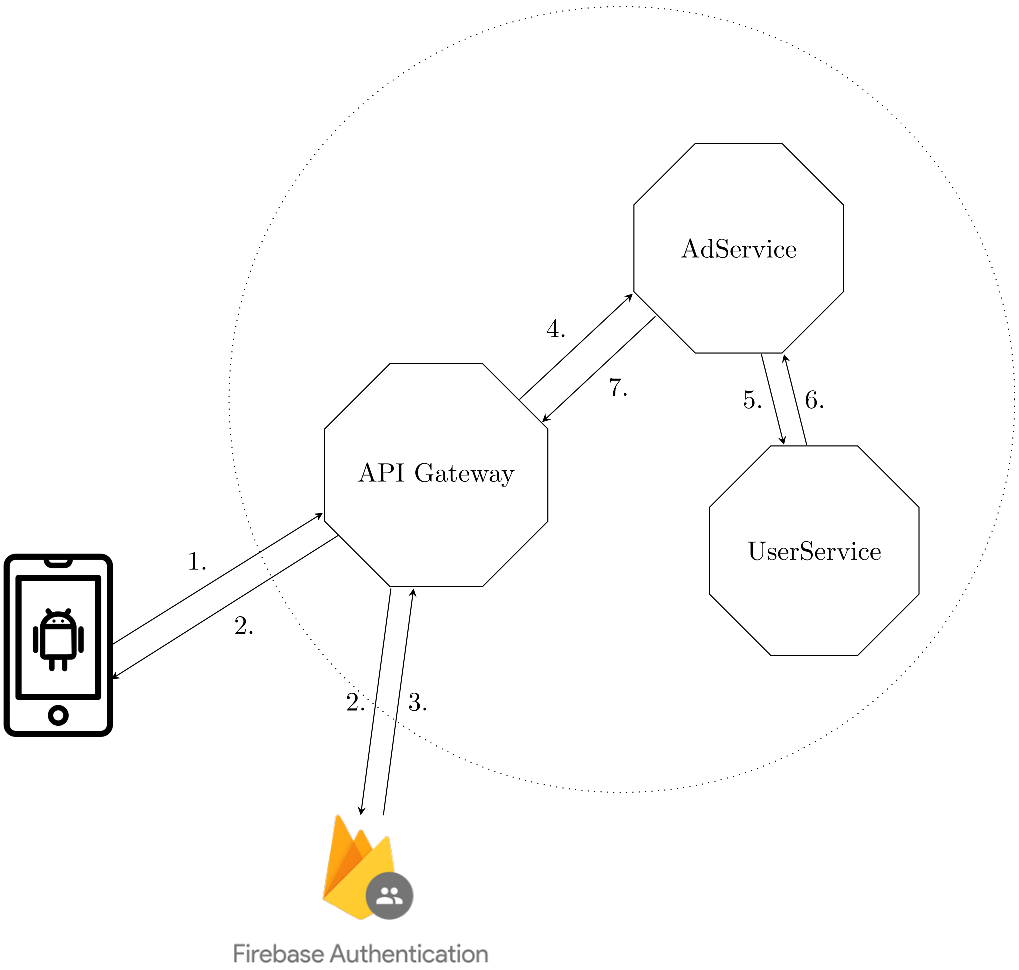

Figure 6.1: Structure of the deployment

vided by firebase. If the token is invalid, the client has to retrieve a new token an start from the beginning. When the client provided an 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 the API Gateway and the AdService have to present a valid certificate signed by a trusted CA.

3. The request from the client is then processed on the “AdService”. Since the “AdService” needs information about the users which are the owners of the ads, it needs to communicate with the “UserService”. The communication between the “UserService” and the “AdService” is also protected using mTLS.
4. When the “UserService” processed the request from the “AdService”, it responds with the expected result. The response does not require to present the certificates again, since TCP connection between the “AdService” and the “UserService” is

still present.

5. The “AdService” then processes the response from the “UserService” and sends its request to the API Gateway, again using the opened TCP connection.
6. The API Gateway forwards the response from the “AdService” to the Android App. This connection is still secured using TLS and not mTLS. It would be possible to secure the communication between the API Gateway and the client using certificate. This mechanism is called Client Certificate Authentication.
7. The App can now process the response and present the requested information to the user.

6.3 Workflow using self signed JWT

References

Literature

[1] Wajjakkara Kankanamge Anthony Nuwan Dias and Prabath Siriwardena. *Microservices Security in Action*. Simon and Schuster, 2020 (cit. on pp. 2, 7, 8, 11, 13–16).

[2] 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. 3, 4).

[3] 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 p. 9).

[4] Aleksandr Kurbatov et al. "Design and implementation of secure communication between microservices" (2021) (cit. on p. 9).

[5] Sam Newman. *Building microservices*. "O'Reilly Media, Inc.", 2021 (cit. on pp. 2–4).

[6] Arnis Parsovs. "Practical issues with TLS client certificate authentication". *Cryptology ePrint Archive* (2013) (cit. on pp. 12–14).

[7] Sean Turner. "Transport Layer Security". *IEEE Internet Computing* 18.6 (2014), pp. 60–63 (cit. on pp. 8, 9).

[8] 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. 16).

Online sources

[9] Auth0. *JWT Documentation*. URL: https://jwt.io (visited on 10/27/2021) (cit. on pp. 7, 8).

[10] Martin Fowler. *Miroservice Premium*. May 2015. URL: https://martinfowler.com/bliki/MicroservicePremium.html (visited on 11/29/2010) (cit. on p. 5).

[11] *RFC2459*. URL: hhttps://www.ietf.org/rfc/rfc2459 (visited on 12/29/2021) (cit. on pp. 6, 7).

[12] *RFC7519*. URL: https://datatracker.ietf.org/doc/html/rfc7519 (visited on 10/27/2021) (cit. on p. 8).

[13] *The Beauty of SSL-Handshake*. URL: https://medium.com/@vraghuvaran123/the-beauty-of-ssl-handshake-4286afa543cf (visited on 01/24/2022) (cit. on p. 12).