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

Ekaterina Shmeleva

How Microservices are Changing the Security Landscape

Master's Thesis
Espoo, November 23, 2020

Supervisor: Professor Tuomas Aura, Aalto University
Advisor: Jacopo Bufalino, M.Sc. (Tech.)

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

ABSTRACT OF MASTER'S THESIS

Author:	Ekaterina Shmeleva		
Title:	How Microservices are Changing the Security Landscape		
Date:	November 23, 2020	**Pages:**	vi + 67
Major:	Security and Cloud Computing	**Code:**	SCI3084
Supervisor:	Professor Tuomas Aura, Aalto University		
Advisor:	Jacopo Bufalino, M.Sc. (Tech.)		

The microservice architecture is an architectural style that structures an application as a collection of fine-grained, self-contained, single-purpose, independently deployable services. Being a young architecture style and a still-evolving one, all aspects of the microservice architecture have not yet been thoroughly analysed in academic literature, especially compared to the fair amount of professional literature that exists on the subject. Hence, the grey literature provides a valuable resource for understanding the microservice architecture and gaining insight into current practices.

Practitioners adopt the microservice architecture to tackle the problems of the monolithic architecture, including security issues. However, the microservice architecture is not a silver bullet and brings its own challenges. Adopting the microservice architecture changes the way security needs to be approached. Microservices have very particular security needs, different from those of a monolithic application, that must be accommodated. This thesis explores these needs and looks into strategies for satisfying them.

Both the edge of the microservice application and the communication between microservices within the application need to be secured. Securing the application at the edge should not cause developers to downplay the importance of securing each microservice at the service-level and working towards adopting zero-trust security principles, which evidently gain popularity in the industry. In the thesis, we discuss end-user and service-to-service access control both at the edge of the deployment and the edge of the service.

Finally, we describe the first step of the incremental process of migrating a monolithic application securely to microservices. We apply the strangler fig migration pattern and extract the identity microservice from the monolith. We evaluate the security of the resulting architecture based on the discoveries presented in the earlier chapters of the thesis.

Keywords:	Microservices, Security, Access Control, Zero Trust, Trust Engineering, DevSecOps
Language:	English

Abbreviations and Acronyms

ABAC	Attribute-Based Access Control
AC	Access Control
ACE	Access Control Entry
ACL	Access-Control List
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
CA	Certificate Authority
CBAC	Capability-Based Access Control
CD	Continuous Deployment
CDN	Content Delivery Network
CI	Continuous Integration
CSRF/XSRF	Cross-Site Request Forgery
DAC	Discretionary Access Control
DDD	Domain-Driven Design
DDoS	Distributed Denial-of-Service
DMZ	Demilitarised Zone
DevOps	Development and Operations
DoS	Denial-of-Service
gRPC	gRPC Remote Procedure Calls
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
IP	Internet Protocol
IPC	Inter-Process Communication
IaC	Infrastructure as Code
JOSE	JavaScript Object Signing and Encryption
JSON	JavaScript Object Notation
JWA	JSON Web Algorithms
JWKS	JSON Web Key Set

JWS	JSON Web Signature
JWT	JSON Web Token
L3	Level 3
L4	Level 4
L7	Level 7
mTLS	Mutual TLS
MAC	Message Authentication Code
MQ	Message Queue
MVC	Model-View-Controller
NAT	Network Address Translation
NIST	National Institute of Standards and Technology
OASIS	Organization for the Advancement of Structured Information Standards
OPA	Open Policy Agent
ORM	Object-Relational Mapping
OSI	Open Systems Interconnection
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PRP	Policy Retrieval Point
Pub/Sub	Publish/Subscribe
RBAC	Role-Based Access Control
REST	Representational State Transfer
RFC	Request for Comments
SEO	Search Engine Optimisation
SPDL	Security Policy Definition Language
SSL	Secure Sockets Layer
SSO	Single Sign-On
STOMP	Simple (Streaming) Text Oriented Message Protocol
STS	Security Token Service
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UUID	Universally Unique IDentifier
VM	Virtual Machine
XACML	eXtensible Access Control Markup Language
XML	Extensible Markup Language

Contents

Chapter 1

Introduction

The microservice architecture is an architectural style inspired by the principles of domain-driven design (DDD), particularly by the concept of bounded context [60]. A microservice is a fine-grained, self-contained, single-purpose, independently deployable unit [61]. Together, microservices form a logically complete microservice application.

Practitioners adopt the microservice architecture to encourage team autonomy and ownership, reduce time to market, scale up and down cost-effectively, enable DevOps, and embrace new technologies. Apart from these real benefits, following the mainstream is a common motivation for adopting the microservice architecture [50, 74].

Tech giants like Netflix and Lyft are not only building their products on top of the microservice architecture but also impact the industry by sharing the insights of how they build their products in blog posts, white papers, and public talks as well as by making their technologies, such as the following, available as open-source:

- Netflix went open source with Zuul, a gateway service that provides dynamic routing, monitoring, resiliency, and security, and several other technologies that power microservices[1].
- Lyft is a company behind Envoy, an open-source edge and service proxy, which works with the Istio and Consul Connect service meshes [16].

However, despite the strong industry presence, all aspects of the microservice architecture have not yet been thoroughly analysed in academic literature. The existing academic research on the topic mainly comprises secondary studies and surveys. At the same time, there is a wealth of grey

[1]Netflix Open Source Software Center. URL: `https://netflix.github.io/` (visited on 10/10/2020)

literature on microservices [68]. This gap between academic research and the industrial grey literature becomes even more pronounced when the topic of microservices is narrowed down. There is even less thorough academic research that focuses specifically on the topic of microservice security. At the same time, microservices have particular security needs, different from those of a monolithic application, that must be satisfied.

1.1 Scope, Aim and Objectives

The scope of the thesis is microservice security or, more precisely, the particular security needs of microservices and industry best practices for meeting those needs, and possible gaps and unsolved problems. The thesis focuses on architectural choices and trade-offs, as opposed to details of implementation.

The aim is to deepen understanding of the challenges and opportunities for microservice security. The thesis has two objectives:

1. The first objective is to undertake an exploratory study to highlight the challenges for security posed by adopting the microservice architecture, explore architectural patterns and strategies that address these challenges, and reveal other, possibly unsolved problems.

2. The second objective focuses on refactoring a real-life monolithic application to microservices and assessing the results from the perspective of the theoretical foundation established in Objective 1.

1.2 Outline

The rest of the thesis is organised as follows. Chapter 2 defines monolithic and microservice architectures, discusses the benefits and drawbacks of the two architectural styles, and presents relevant architectural patterns and access control models. Chapter 3 lists the specific challenges of securing microservices. Chapter 4 discusses securing public traffic to a microservice deployment, particularly enforcing security policies at the edge of the deployment and restricting access to underlying services which the consumer should not access directly. Chapter 5 defines and justifies the concept of zero trust security and discusses securing private traffic within a microservice deployment by encrypting the traffic and verifying claims that microservices make about both the end-user and themselves. Chapter 6 describes the process of migrating a monolithic application to the microservice architecture with a focus on the security aspects of the new architecture and gives an

evaluation of the process and results. Finally, Chapter 7 summarises the work and discusses open problems and future work.

Chapter 2

Background

This chapter gives background on microservices as an architectural style. It begins by assessing the gains and costs of monolithic and microservice architectures. After this, the chapter discusses the challenges of securing microservices found in academic and grey literature. It continues with a discussion of architectural patterns for securing microservices and their implementations. Afterwards, the chapter describes and compares two types of access tokens. Finally, it ends with a presentation of several access control models.

2.1 Monolith to Microservices

This section defines monolithic and microservice architectures, assesses the gains and costs of adopting one or the other, and identifies the motivations for migrating a monolith to microservices.

2.1.1 Everything Wrong with Monoliths

Even though a monolithic application can comprise several parts, such as presentation, business logic and data tiers in a three-tier architecture, it is built and deployed as a single unit [5]. While a monolithic application is a suitable choice for a smaller application, it suffers from several issues that become more evident as the monolith grows:

A. Codebase complexity

As a monolith grows, the complexity of the codebase increases, while the quality of the codebase declines over time. This results in challenges

for understanding and navigating the codebase and ultimately leads to more security bugs and vulnerabilities [15]. However, sound development practices such as producing clean and maintainable code, conducting peer reviews, and writing documentation help foster security by reducing the complexity of a large codebase [64].

B. Interconnectivity and interdependency

Since the parts of a monolith are interconnected and interdependent, a single change in a monolith might start a chain reaction of unrelated test failures for non-obvious reasons, thus requiring additional changes that must be coordinated across different parts of the monolith [81]. A single change in one part of the monolith might cause a bug or vulnerability in another part of the monolith, which may remain undetected by tests [5].

C. Steep learning curve

It might take months for a new developer joining a team to become effective and comfortable with making their first non-trivial contribution. For example, before Shopify decomposed their monolithic system into microservices, a new developer on the shipping team would have also needed to learn how orders are created, how payments are processed, and more [81].

D. "Dependency hell"

A large number of dependencies makes it challenging to update a component if there are other parts of the system that directly or indirectly depend on an incompatible older version of the component. An update may result in an inconsistent system that does not build, run, or behave as intended [15].

E. Slow development cycles and considerable downtimes

Even the smallest change in a single component of a monolith requires that the whole application is rebuilt, retested and redeployed, which might result in considerable downtimes [5, 15, 72]. Moreover, a large codebase slows down an IDE, which further slows down the process of development.

F. One-size-fits-all configuration

Individual components of a monolith might have varying resource and technology requirements. As a result, while designing and configuring a monolith, the developer must settle for a solution which might be sub-optimal with respect to every component [15].

G. Limited scalability

Different components of a monolith might have different scaling requirements. It could be that some components receive more inbound requests than others. To deal with an increase in traffic, the entire monolithic application needs to be replicated. This wastes resources and limits scalability [15, 72].

H. Technology lock-in

It is not easy to adopt new technologies in a monolith and evolve its components independently [15, 72]. For example, it took a team of four full-time engineers plus additional volunteers a year and a half to upgrade monolithic GitHub from Rails 3.2 to Rails 5.2 [77].

2.1.2 The "Why" of the Microservice Architecture

A microservice application is comprised of fine-grained, self-contained, single-purpose, independently deployable units [61]. Each microservice runs within its own process and includes everything needed to operate independently, such as its own persistent storage [5, 15, 60].

Within a microservice deployment, microservices interact with one another using IPC protocols such as HTTP, message-based AMQP and STOMP, or binary TCP and UDP [76]. While the choice of the appropriate protocol depends on the nature of each microservice, microservices must be aware of what protocols they can use to interact with other microservices [60].

Richardson lists a few different microservice deployment patterns, which are *Deploying a Service as a Virtual Machine*, *Deploying a Service as a Container*, and *Serverless Deployment* [61]. A container-based microservice deployment is a common approach to streamline the DevOps process [29]. Containers are more lightweight than VMs but do not have the significant limitations of a serverless deployment. The figures from a 2018 report by Datadog show a steady increase in the adoption of container technologies such as Docker [1]. To limit the scope of the thesis, hereinafter, it is assumed that each service is packaged as a container image, and each service instance

runs in its own container. The term "container" refers to a Docker container unless otherwise specified.

The microservice architecture provides the means for overcoming the disadvantages of the monolithic architecture described earlier in this chapter. Below are the respective advantages and, if applicable, disadvantages of the microservice architecture:

A. Failure isolation and independent life cycles

The narrow focus of each microservice helps to keep the codebase relatively small and essentially narrow the search area for a bug [15]. Furthermore, every microservice can evolve at its pace, have its own repository and continuous integration and deployment (CI/CD) pipeline, and be deployed in a different cloud provider. However, whilst the microservice architecture fosters simplicity and eases maintainability of individual components, it introduces additional network complexity, especially as the number of microservices grows. As discussed further in Chapter 3, this complexity brings additional challenges for, inter alia, security.

B. Independence

Microservices have clear boundaries; they are independently deployable and operationally independent of one another. This independence minimises the risk of side effects of making a code change in the microservice on the others [5, 9], and it allows the developer to run tests on the microservice in isolation from the rest of the system [15] and to run more comprehensive regression tests than in a larger monolithic application [59]. However, to achieve this independence, each microservice must adhere to a well-defined, versioned API contract and support the previous versions until no other microservice relies on those particular versions of the API contract.

C. Swift learning

The developer can make a non-trivial code change without knowing how other microservices are implemented. In contrast to the previous example with Shopify, if shippings, orders and payments are each handled by a different microservice, the developer on the shipping team would not have to take time to understand the internals of the two other microservices.

D. Gradual rollouts and continuous updates

A new version of a microservice can be rolled out gradually, and two or more versions of the same microservice can exist in parallel. This way, other microservices can gradually move to a newer version [15].

E. Brief redeployment downtimes and seamless rollouts

Changing a single microservice requires that this microservice alone is rebuilt and redeployed, as opposed to the entire system. Due to the small size of a microservice, this results in brief redeployment downtimes [15] and, in case of failure, only a part of the whole system will be affected [5]. There are a few commonly used deployment patterns for updating a running microservice to a newer version that typically consider achieving zero downtime, minimising the impact of deployment incidents and failures, and yielding reliable, predictable, and repeatable deployments [4]:

- In a *rolling update* deployment, the new version is gradually rolled out to subsets of running microservice instances until it has completely replaced the old version. A slow, gradual rollback is the main consideration of a rolling update deployment.
- In a *blue/green* deployment, the new version is released alongside the old version, and traffic is switched to the new version once it has been tested. An instant rollback is the key benefit of a blue-green deployment; however, it requires maintaining the blue and green environments simultaneously, which carries cost implications.

F. Service-specific configuration and freedom to choose the right technology

The microservice architecture is typically a heterogeneous, polyglot architecture. It promotes different microservices in a system to pick the technology stack depending on the nature and needs of the microservice as well as other factors such as price and familiarity with the technology [60]. Moreover, each microservice can have a separate environment configuration based on its specific needs [15].

G. Cost-effective scalability

Typically, different parts of a complex application have different scaling requirements. Microservices can scale at different rates to respond to varying load conditions. Instead of adding and removing instances of the

entire application, only the microservices that are under heavy load can be scaled up [5, 15]. Based on performance tests, Villamizar et al. conclude that the adoption of the microservice architecture can help reduce up to 13 percent of infrastructure costs [79]. By deploying microservices within a serverless architecture with AWS Lambda, Google Cloud Functions or Azure Functions, the infrastructure costs can be reduced by up to 77 percent, according to Villamizar et al. [79], or by 90 percent or more, according to Singleton [65].

H. No technology lock-in

Developers are free to choose the suitable technology stack to implement a microservice [9, 15] and to replace the existing microservice with a new microservice built with a different technology stack, as long as the new microservice adheres to the same API contract as the old one [5, 9, 82]. The only constraints are imposed by the service-to-service communication mechanisms [15].

2.2 Microservice Patterns

One approach to building a microservice application is to use a direct client-to-microservice communication architecture in which each microservice provides a public entry point that clients can access directly, without middleboxes [61, 76]. However, while suitable for a small system, the direct client-to-microservice architecture is rarely used in practice [61] and considered an anti-pattern as the number of microservices increases [73]. This is due to the following reasons:

Increase in the number of round trips

Microservices typically provide fine-grained APIs. Should the consuming application need to make multiple requests to multiple microservice APIs as part of one higher-level operation, the total number of round trips between the consuming application and the microservice application increases [61, 76]. This results in higher latency, especially if the servers are geographically remote from the client.

Coupling between consumers and microservices

The client application directly depends on the microservice APIs and the internal architecture of the microservice application, which compli-

cates the client application code and increases the cost of introducing breaking changes to the API contracts and the internal architecture, for example, while refactoring microservices [61, 76]. The rawness of microservice APIs caused by prioritising scalability and robustness above the API design best practices also does not make them ideal for external exposure [67].

Client-unfriendly protocols used by the services

Microservices may need to use IPC mechanisms that cannot be easily consumed by the client application. These include HTTP-based gRPC or message-based AMQP and STOMP. Without protocol translation, the choice of IPC mechanisms is limited to client- and firewall-friendly ones such as HTTP-based REST and WebSockets.

Poor defence-in-depth

Regardless of the size of the application, the direct client-to-microservice communication has security limitations: without a gatekeeper, the internal network of the application is exposed to unauthenticated or unauthorised requests [61].

This section discusses three architectural patterns that help address the above-listed limitations, including their description, use cases, benefits and drawbacks, and implementations that are in use today.

2.2.1 API Gateway Pattern

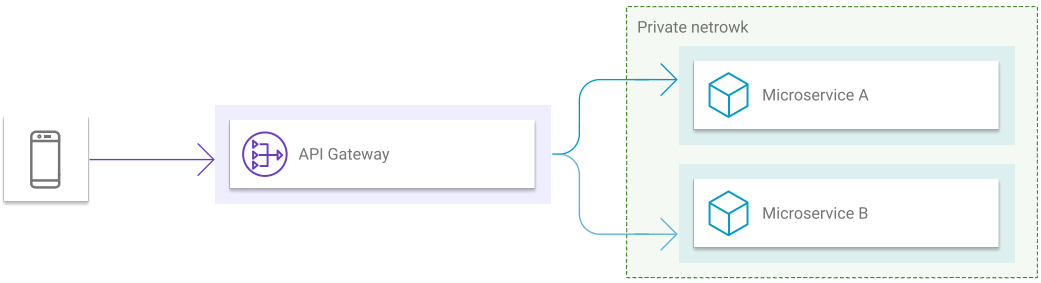

Figure 2.1: API gateway pattern

The *API gateway* pattern is the most common microservice architectural patterns found in microservices, as evidenced by a study of open-source

projects [46]. An ingress API gateway is a service that typically sits behind the firewall in the data centre and acts as a single point of entry into an application from the internet outside the firewall. It is responsible for request routing and composition, protocol translation, and handling cross-cutting concerns [61]:

Request routing and composition

An API gateway acts primarily as a reverse proxy, routing requests from the clients to the microservices behind it. Upon receiving a request from the client, an API gateway forwards it to the designated microservice based on the client IP address, server port number, request headers, or requested URL, and returns the response from the microservice to the client.

However, unlike a reverse proxy, an API gateway can also act as an aggregator and not only route a request from the client to the corresponding microservice but also invoke multiple microservices and aggregate the results into a single response. API composition addresses two of the above-mentioned limitations of the direct client-to-microservice communication architecture.

Firstly, providing the clients with a coarse-grained API reduces the number of round trips between the consuming application and the microservice application.

Secondly, request routing and composition at the API gateway decouples the clients from the microservices. However, request aggregation comes with a risk of coupling between the gateway and microservices.

A variation of the API gateway pattern is the *backend for frontend* pattern. It suggests having multiple API gateways to accommodate the unique needs of every client application and to simplify the client applications by moving the aggregation logic into an application-specific API gateway accessed through its own API.

Protocol and data interchange format translation

Microservices interact with one another using a large variety of IPC mechanisms. Depending on the nature of each microservice and whether the interaction is synchronous or asynchronous, microservices can either use synchronous IPC mechanisms, such as HTTP-based REST and gRPC [25], or asynchronous IPC mechanisms, such as HTTP-based WebSockets and message-based AMQP and STOMP. Likewise, microservices can use a wide range of data interchange formats, such as text-based

JSON and XML or binary Protocol Buffers [57]. An API gateway can translate between the protocols used internally by the microservices and client- and firewall-friendly protocols such as HTTP and WebSockets [61, 76].

Offloading cross-cutting concerns

In addition to request routing, composition, and protocol translation, an API gateway often handles cross-cutting concerns that are shared across multiple microservices. These include authentication and authorisation, rate limiting, throttling, retry policies, circuit breaking, logging and monitoring, caching, and other concerns.

Abstracting cross-cutting concerns and implementing them in a single place such as an API gateway lowers the complexity of the application and improves its maintainability.

API gateways can also be layered in front of each other and implement different operational concerns for a cleaner separation of concerns [63]. However, this incurs an increase in latency due to one or more extra network hops.

While the API gateway pattern resolves many of the above-listed issues of a direct client-to-microservices architecture, it also creates a potential development bottleneck since routing and aggregation rules may have to be updated following a change to the microservice APIs before the client can get the update. Likewise, the API gateway is a potential single point of failure, and it must be able to handle the load and scale up appropriately. Finally, having an API gateway creates an additional network hop between the client and microservice.

The availability of technologies for implementing API gateway predates the rise of the microservice architecture. Below is a reasonably inclusive list of widely used L4 and L7 proxies and API gateways:

- L4 and L7 proxies *NGINX*, *HAProxy*, *Envoy*, *Traefik*,
- API gateways *Zuul*, *Kong*, *Tyk*, *Ocelot*, *Ambassador*,
- API gateways as a service *Amazon API Gateway* and *Azure API Management*.

2.2.2 Sidecar Pattern

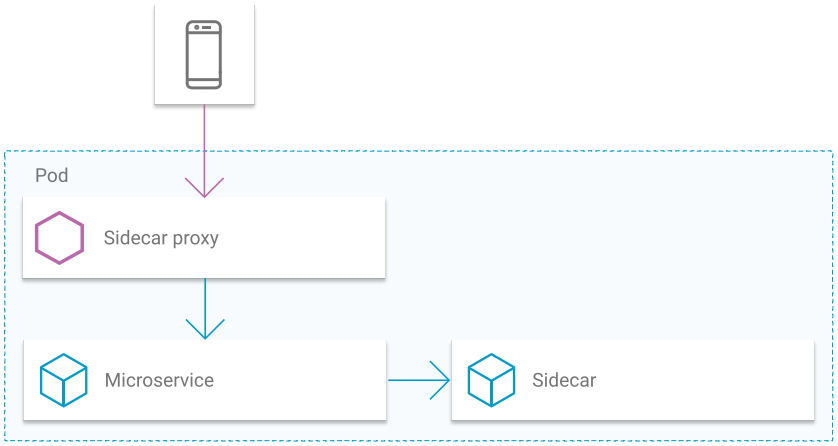

Figure 2.2: Sidecar and sidecar proxy patterns

A sidecar is a container that runs alongside another container, which runs the main microservice and is sometimes referred to as the main car. The main car can have multiple sidecars [67]. The containers are co-located and co-scheduled on the same physical or virtual machine via an atomic group of containers such as a Kubernetes pod [8]. A pod is the smallest unit of deployment in Kubernetes that encapsulates one or more containers [38]. Every container in a pod shares network and storage resources and the same lifecycle, i.e., a sidecar is always created and retired alongside the main car.

The purpose of a sidecar is to improve or augment the functionality of the main car. For example, the sidecar can act as a service proxy to the main car and implement rate limiting, throttling, logging, tracing, protocol translation, security, and other cross-cutting concerns. Several sidecar proxy implementations exist, with Envoy Proxy being one of the dominant ones [16].

The sidecar pattern helps to reduce development effort by moving shared functionality out of each microservice and into a sidecar [60]. This also allows for a clearer separation of concerns between the core functionality of a microservice and common functionality shared by microservices [63].

However, running an extra sidecar container within each pod increases the resource cost of a microservice deployment, which becomes problematic as the number of microservices and sidecar proxies grows [63].

Containers within a pod can communicate with each other using localhost or standard IPC mechanisms such as named pipes, message queues, or shared memory [38]. While these are faster than remote communication between containers that are not co-located and co-scheduled, communication between

containers in the same pod is still slower than language-level function and method calls [8].

2.2.3 Service Mesh Pattern

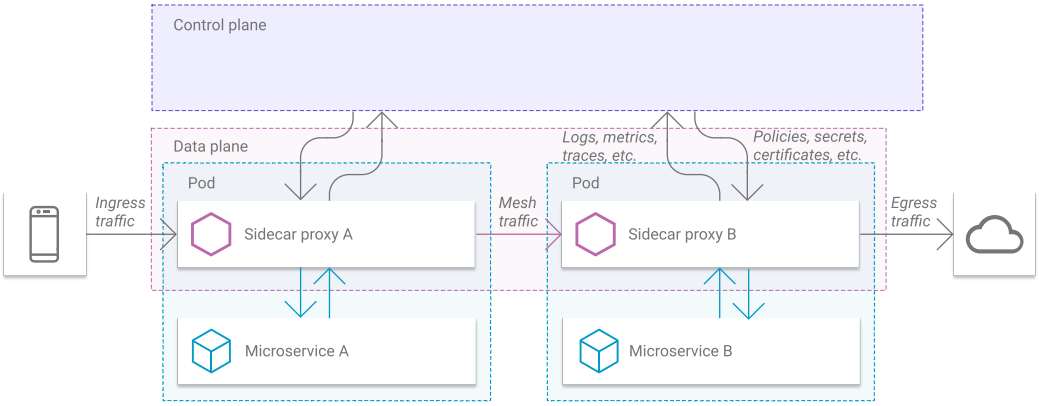

Figure 2.3: Service mesh pattern

A service mesh is a configurable, low-latency infrastructure layer that mediates ingress and egress traffic that flows to and from each microservice, secures the traffic with a network security protocol like TLS, and handles various cross-cutting concerns, such as distributed tracing, circuit breaking, service discovery, and rule-based load balancing and traffic routing [61, 76].

A service mesh is logically split into a data plane and a control plane [76]:

- The *data plane* is typically comprised of a network of sidecar proxies that run in each pod alongside the main car. Each sidecar proxy routes or proxies ingress and egress traffic to and from the microservice to which it is attached.
- The *control plane* is a set of APIs and tools used to manage and configure the proxies across the service mesh.

Table 2.1 summarises four widely used open-source service mesh implementations based on the technologies they employ. All the service meshes listed in the table support TLS, HTTP/1.1, HTTP/2 and gRPC protocols and mTLS for mutual authentication. All except Linkerd2 also provide built-in authorisation features [11, 30, 40, 43].

	Sidecar proxy	Ingress controller	Platform
Istio [30]	Envoy	Envoy	Kubernetes
Linkerd2 [43]	Native `linkerd2-proxy`	Any	Kubernetes
Kuma [40]	Envoy	Any	Kubernetes
Consul Connect [11]	Native, Envoy, NGINX, HAProxy	Envoy, Ambassador	Kubernetes, Nomad

Table 2.1: Comparing service mesh implementations

2.3 Access Tokens

Various types of access tokens are commonly used in the microservice architecture for tasks such as authenticating users and services and securely transmitting information about the user or service. The specifics of how access tokens can be used in these tasks are discussed further in Chapters 4 and 5. This section presents an overview of two types of access tokens, opaque and transparent tokens.

Opaque tokens

An *opaque token*, also called *reference token*, is a handle that refers to a record stored in the persistent storage of the issuing service. It is typically represented by a plain UUID, as defined in RFC 4122 [41]. The UUID is a sequence of 128 bits represented as a lower-case hexadecimal string consisting of a group of 8 digits followed by three groups of 4 digits followed by a group of 12 digits, with the groups separated by hyphens. The UUID offers no security features for verifying the authorship of the token, nor can it explicitly contain information about the user or service. To validate an opaque UUID token and retrieve information about the user or service, the recipient of the token must invoke the service that issued the token, which reduces both performance — due to one or more extra network hops to the issuing service — and availability as the recipient of the token cannot proceed with the request before the token has been validated.

Transparent tokens

A *transparent token*, also called *self-contained token*, on the other hand, is capable of containing information about the user which the recipient of the token can read without invoking the issuing service. It is typically expressed as a JSON Web Token (JWT).

JWT is a compact and self-contained claims representation format for transmitting information between parties as a JSON object in a cryptographically safe manner, as defined in RFC 7519 [33]. A claim is a statement about a subject represented as a name-value pair and asserted by some entity such as a Security Token Service (STS).

A set of claims can be digitally signed or integrity protected and encrypted. A digitally signed JWT consists of three Base64-encoded sections separated by two periods:

- *Header* is a Base64-encoded JSON object that typically contains the type of the token and the signing algorithm. The type of the token is specified in the "typ" parameter. It is recommended that its value be set to "JWT". The signing algorithm is specified in the "alg" parameter, as defined in the JSON Web Algorithms (JWA) specification [31].
- *Payload* is a Base64-encoded JSON object that that contains a set of claims. RFC 7519 defines a set of predefined claims, referred to as registered claims, which encode identity and metadata and are recommended for interoperability. These include "iss" (issuer), "exp" (expiration time), "sub" (subject), and "aud" (audience).
- *Signature* provides integrity protection and source authentication. JSON Web Signature (JWS) is defined in RFC 7515 [32].

JWTs are not dependant on any specific technology stack, meaning that they can be used in the polyglot microservice architecture since many implementations of JWT exist.

Table 2.2 presents a comparison of UUID tokens and digitally signed JWTs. It reflects the current industry preference for JWTs, which are considered to have more sophisticated security features and offer greater flexibility in striking balance between security and performance.

	UUID tokens	Digitally signed JWTs
Performance	Information about the user or service must be retrieved from the issuing service	Explicitly contain information about the end-user
Expiration	Only if implemented in the issuing service	Built-in
Revocation	Only if implemented in the issuing service	Only if implemented in the issuing service AND the recipient validates the token by invoking the issuing service
Integrity	No	Built-in

Table 2.2: Comparing UUID tokens and JWTs

2.4 Access Control Models

Access control (AC) is the process of permitting or restricting access to resources only to privileged entities based on a defined AC policy. An AC policy is a set of rules that define the conditions under which access may take place, which can be formally presented as an AC model. An AC model links an AC policy and mechanism [27]. An AC mechanism is an implementation of an AC model. The primary role of an AC mechanism is to receive an access request, reach an access decision, and enforce this decision [28]. An AC mechanism might include all or some of the following integrated or separate core components:

- *policy administration point* (PAP), where policy administrators and developers define AC policies [17, 54];
- *policy decision point* (PDP), where policy decisions are made [17, 54, 80];
- *policy enforcement point* (PEP), where the policy decisions are enforced [17, 54, 80];
- *policy information point* (PIP), from where the PDP can load external data [17, 80];
- *policy retrieval point* (PRP), from where the PDP can load AC policies [17, 80].

2.4.1 Access Control Lists

An access control list (ACL) is a list of access control entries (ACE) attached to a resource. Each ACE identifies the user and specifies access modes granted to the user.

2.4.2 Role-Based Access Control

Role-based access control (RBAC), as formalised by Ferraiolo and Kuhn [19], is a policy-independent identity-based access control model that exploits the notions of users, roles and permissions. RBAC establishes many-to-many role-permission, user-role, and role-role relations. Each user is assigned one or more roles, and each role is assigned one or more permissions. The 1993 NIST study by Ferraiolo et al. concludes that role-permission relations tend to be more permanent than user-role relations [18]. Roles can have inheritance, mutual exclusion, and other relations. Through these relations, RBAC directly supports the principles of least privilege, separation of duties, and data abstraction [62]. *Least privilege* ensures that users and roles are assigned the absolute minimum permissions necessary. *Separation* or *segregation of duties* ensures that no user or role is assigned all necessary permissions to perform a malicious act. *Data abstraction* ensures the reduction of the specific read, write, execute permissions to abstract permissions.

2.4.3 Attribute-Based Access Control

The RBAC model has several known limitations. Firstly, it provides coarse-grained AC while many applications require finer-grained AC [58]. Secondly, it makes decisions based on the statically assigned roles and ignores the dynamically changing context. Attribute-based access control (ABAC) overcomes these limitations of RBAC. It leverages contextual attributes associated with the requesting user, requested object and current environment [58]. Policy rules are specified in the form of Boolean expressions consisting of contextual attributes. Access is granted if the expression is true and denied otherwise.

ABAC can be seen as a generalisation of ACL and RBAC models that employ the identity and role attributes accordingly [28]. ABAC is typically more complex than RBAC in terms of policy review; hence ensuring the validity of a policy and reviewing or modifying user permissions may not be practically feasible [12]. As a result, there have been multiple efforts to integrate RBAC and ABAC models to combine the advantages offered by each model [12, 39, 58].

ABAC policies can be implemented with the Open Policy Agent and Speedle+ AC engines as well as with several AC engines that implement the XACML standard:

Open Policy Agent

Open Policy Agent (OPA) is an open-source, lightweight, general-purpose policy engine that allows to define fine-grained AC policies and offload the enforcement of these policies to a different location. That is, OPA decouples policy decision making and enforcement [54].

OPA policies are written in Rego, its own high-level declarative language. Rego supports variables; basic arithmetic, comparison, and logical expressions; aggregate functions for summarising complex types such as objects, arrays, and sets; and rules. Rules are if-then logic statements that can either be partial or complete. Partial rules generate a set of values and assign it to a variable. Complete rules assign a single value to a variable and can be regarded as a special case of partial rules. In addition to ABAC policies, Rego is suitable for defining ACL and RBAC policies.

OPA can be deployed as a Docker container. The OPA server exposes a REST API that the PEP can query to check authorisation.

OPA supports integration with Kubernetes, Istio, Kafka, Terraform and multiple other projects.

Speedle+

Speedle+ is an open-source, general-purpose policy engine. Similarly to OPA, it allows to define fine-grained AC policies and decouple decision making and enforcement of these policies [69].

Speedle+ policies are written in its own Security Policy Definition Language (SPDL), which can describe both RBAC and ABAC policies.

Speedle+ supports integration with Docker, Kubernetes and Istio.

XACML

eXtensible Access Control Markup Language (XACML) is an open standard for expressing security policies developed by the Organization for the Advancement of Structured Information Standards (OASIS) [17]. The XACML standard introduces an XML-based policy language and an XML-based schema for authorisation requests and responses. The latest version of the standard, XACML 3.0, was released in 2013.

2.4.4 Capability-Based Access Control

A capability is an unforgeable handle for a resource coupled with a set of permissions to access that resource. As opposed to RBAC and ABAC models, capability-based access control (CBAC) model is a non-identity-based AC model. Instead of conveying the identity of a user, from which the permissions available to the particular user can be determined, a capability directly communicates the permissions [45]. The use of capability tokens, which are scoped to an individual resource, provides the finest-grained AC among the three AC models, thus adhering tightly to the principle of least privilege. In addition, capabilities decrease the risk of a confused deputy attack [26].

For accountability reasons, an application may need to authenticate the user in addition to authorising the request. Capabilities can be combined with identities either by tying the identity of the user to each capability token or by using a separate authentication mechanism to identify the user in addition to requiring a capability token to authorise the request. However, the first approach is only suitable when a capability token is short-lived and intended for a single user. Sharing such capability token with other users would allow them to impersonate the user whose identity is associated with the capability token.

CBAC is intended for fine-grained authorisation and delegation of access to individual resources in decentralised, distributed systems. However, there is no evidence of widespread use of CBAC in web API security, despite the steady increase in the adoption of microservices which are distributed by nature.

	RBAC	ABAC	CBAC
Fine-grained	No	Yes	Yes
Context-aware	No	Yes	Yes
Easily reviewable	Yes	No	No
Easily modifiable	Yes	No	*NA*
Identity-based	Yes	Yes	No

Table 2.3: Comparing AC models

Chapter 3

Challenges in Securing Microservices

In many aspects, the security of the microservice architecture is no different from that of the monolithic architecture. A microservice application is susceptible to the same set of attacks as traditional monolithic applications, including L3/L4 attacks such as network-layer denial-of-service (DoS) and distributed denial-of-service (DDoS) attacks and L7 attacks such as cross-site scripting (XSS), cross-site request forgery (CSRF/XSRF), SQL injections, and application-layer DoS/DDoS attacks. However, the distributed nature of the microservice architecture poses specific security challenges that do not exist or are greatly attenuated in the monolithic architecture.

A significant portion of academic research on microservice security comprises works that aim to identify the challenges of securing microservices. To summarise and systematise these security challenges, microservice security is commonly divided into layers:

- Yarygina and Bagge identify six progressive layers inspired by the OSI model. These are the hardware, virtualisation, cloud, communication, application and orchestration layers [83].
- Yu et al. define four aspects of microservice security: containers, data, permission, and network [84].
- Nehme et al. divide their security model into four broad dimensions: the microservices themselves, application architecture, underlying infrastructure, and external interfaces [48].
- Google describes the security of their technical infrastructure in six different progressive layers. These are hardware infrastructure, service

deployment, user identity, storage services, internet communication and operational security [24].

The following chapter gives an overview of the challenges that are specific to the microservice architecture. We omit those challenges that affect both monolithic and microservice applications with equal strength. In terms of the classification introduced by Yarygina and Bagge, we focus on the upper communication, application, and orchestration layers and omit the lower hardware, virtualisation, and cloud layers.

3.1 Protecting Larger Attack Surface

In a monolithic application, components either run on a single process and use language-level function and method calls to invoke one another [67] or use IPC mechanisms, such as files, signals, sockets and pipes, to exchange data among two or more threads in one or more processes [15]. As a result, a monolithic application typically has few entry points.

On the other hand, in a microservice application, microservices communicate over the network. Each microservice exposes one or more entry points to the network, which significantly increases the overall number of entry points and, consequently, expands the attack surface [53, 67]. Since the application as a whole is only as protected as its weakest point, all the entry points must be equally secured [66, 67]. In addition to securing all the entry points, the integrity and confidentiality of the data in transit must be protected [15].

Chapter 4 discusses reducing the attack surface by enforcing perimeter security, and Chapter 5 discusses securing all the entry points by enforcing zero trust security.

3.2 Striking Balance Between Security and Performance

While the security checks in a monolithic application are typically done only once, in a microservice application, they must be done repeatedly at every entry point as the request propagates through multiple microservices. These repetitive and redundant security checks can have a strong negative impact on the performance of the application [60, 67]. Moreover, these security checks may require that microservices send a request to a remote identity service. Remote calls take longer than language-level function and method

calls. This can further increase latency, resulting in an even more significant performance drop [67].

Alternatively, the security checks can be done once at the edge of the microservice application. As the number of security checks decreases from many to one, the performance of the application improves accordingly. However, this approach assumes that the private network is trusted, which contradicts the principles of zero trust networking and is perceived as an antipattern with the industry shifting towards service mesh solutions [67].

The perceived performance of an application impacts the user experience. Nielsen states that a delay of fewer than 100 milliseconds feels instant to the user, and a delay of between 100 and 1,000 milliseconds is perceptible but does not interrupt the conscious thought process of the user, while a delay of more than 1,000 is likely to cause the user to context-switch [51]. Such long delays are found to reduce user engagement and conversion rates:

- Based on a set of bounce and conversions data from mobile devices, Google predicts that as page load time goes from 1,000 milliseconds to 3,000 milliseconds, the probability of the user leaving increases by 32 percent [2].
- BBC reports that 10 percent of their users leaving for every 1,000-millisecond increase in the page load time [10].
- Pinterest reports a 15 percent increase in the number of visits and sign-ups after a 40-percent reduction in the user-perceived load time [47].

Besides this, low latency becomes increasingly critical in workloads such as intelligent transportation and traffic systems, autonomous vehicles, and virtual and augmented reality. As a result, the trade-off between security and performance should be considered when integrating security into the design [67] and deciding on the appropriate level of service granularity [60].

Chapters 4 and 5 evaluate the ramifications of integrating security into a microservice application for the performance of the application and discuss how they can be overcome.

3.3 Establishing Trust Between Microservices

It only takes one compromised microservice to put the whole microservice deployment in jeopardy. To reduce the trust put on individual microservices and limit the potential damage, zero trust security model can be adopted.

The zero trust principles include protecting the data in transit with TLS, adhering to the principle of least privilege through service-to-service authentication and fine-grained access control, and monitoring service-to-service interactions. Protecting the data in transit and authenticating one microservice to another involves provisioning of keys and certificates to microservices, key revocation, key rotation, and key use monitoring.

3.4 Sharing User Context

All the components of a monolithic application share the same web session and the identity of the end-user is retrieved from it. In comparison, microservices generally do not share resources [1]. Therefore, the identity of the end-user must be explicitly passed between microservices [61, 67].

In a zero trust security model, a microservices must not trust whatever another microservice claims about the end-user. Therefore, the identity of the end-user must be verifiable.

Several methods for passing the identity of the end-user from one microservice to another, along with their benefits and drawbacks, are discussed further in Section 5.1.

3.5 Managing Policies and Secrets

As a best practice for operating containers, the container image has to be immutable for a safe, reliable, and reproducible deployment [6]. Immutability means that the container cannot be modified once it has been built and deployed. As a result, the container configuration must be externalised to dynamically update a list of whitelisted clients, access-control policies, and secrets such as keys and credentials [67]. Moreover, with the dynamic microservice architecture that allows for rapid scaling-up and scaling-down of microservices, enforcing the traditional perimeter security model [2] by statically configuring IP addresses is futile [53].

[1] This is not always the case. For example, the containers in a Kubernetes pod do share network and storage resources.

[2] The perimeter model is explained in Chapter 5.

3.6 Collecting Logs, Metrics, and Distributed Traces

With a higher number of microservices comes complexity, which brings additional challenges for collecting detailed observability data like metrics, distributed traces, and logs, which need to be analysed for malicious, suspicious or anomalous patterns to detect and prevent security breaches and to accelerate security incident investigations. A single request can propagate through multiple microservices, which makes it harder to trace [67]. This becomes even more challenging in applications that embrace the polyglot nature of the microservice architecture. Companies that use the same core set of technologies across all microservices have the most success in integrating distributed tracing [70]. A service mesh also greatly simplifies implementing tracing functionality into a heterogeneous microservice application [70].

3.7 Bringing Diverse Security Expertise to Secure Heterogeneous Microservices

As discussed previously in Chapter 2, the microservice architecture is heterogeneous in nature, and different microservices can use different technology stacks. The shortcoming of a heterogeneous, polyglot architecture is that, without a centralised security team, the responsibility for securing microservices is distributed among many development teams, which requires expertise in security from each team [15, 67]. To limit the responsibility of individual development teams, a hybrid approach can be used, where a single centralised security team collaborates with many development teams [67].

Chapter 2 introduced Open Policy Agent (OPA), which is an open-source project started in 2016 as an effort to unify policy enforcement across a variety of languages and frameworks.

Chapter 4

Edge Security with an API Gateway

In the microservice architecture, edge security, also referred to as perimeter security, often implies securing microservices at the API gateway level. In fact, the API gateway pattern is the most common pattern for securing microservices in a production environment. An ingress API gateway represents the only entry point to a microservice deployment, which handles north-south traffic, also called public traffic, by intercepting all the requests from the public internet before dispatching them to the microservices behind it. Thus, an API gateway can act as a centralised PEP, which enforces authentication, authorisation, and throttling policies and only allows legitimate requests to reach the microservices behind it [66].

However, implementing authentication, authorisation, and throttling in an API gateway by itself is not sufficient to protect the underlying services from unwanted requests originating from outside the microservice deployment. To guarantee that the API gateway alone can access the underlying services and prevent requests coming from outside the microservice deployment from bypassing the API gateway, there must be a firewall, mutual TLS, or both between the API gateway and the underlying services.

Summarising the above, perimeter security comprises enforcing security policies at the edge and restricting access to the underlying services. These two aspects of securing microservices at the edge of a microservice deployment are discussed further in this chapter.

In addition to authentication, authorisation, and throttling, an API gateway would typically implement transport- or application-layer protocols, such as TLS and HTTPS, and DoS/DDoS protection. However, these measures are not specific to microservices and hence are not the focus of this chapter.

4.1 Enforcing Access Control Policies at the Edge of the Deployment

Authentication and authorisation are two closely related yet distinct security mechanisms. Authentication is the process of verifying the identity of the user, while authorisation is the process of verifying whether a specific user is allowed to perform a particular action. This section discusses enforcing end-user authentication and authorisation policies at the edge of the deployment. Here, the term "end-user" does not refer exclusively to human users but also includes other non-human actors that might be making calls to the API from outside the microservice deployment.

As discussed earlier, a monolithic application typically has only a couple of entry points. Only some but not all components of a monolithic application accept requests directly; others are hidden from the outside world and can only be accessed through language-level method or function calls. The requests to a monolithic application are typically funnelled through authentication and authorisation *middleware*. The middleware acts as a centralised policy decision and enforcement point. It intercepts all the requests and either passes them to the next function in the chain or rejects them. Modern frameworks for building internet-connected applications such as *ASP.NET Core* and *Express.js* provide authentication and authorisation middleware as out-of-the-box functionality [3, 78].

From the perspective of the end-user, accessing microservices via an API gateway is no different from accessing a monolithic application: the API gateway hides the inherent complexity and rawness of a microservice deployment from the consuming application and acts as a centralised PEP, conceptually similarly to authentication and authorisation middleware in a monolithic application.

Different consuming applications can authenticate themselves using different authentication schemes. The OpenAPI 3 Specification, an industry standard for describing REST APIs, defines several security schemes which the consuming application may use, including at least Basic authentication; Bearer authentication; API key-based authentication in a header, query string or cookie; OAuth 2.0; and OpenID Connect Discovery [55]. Handling authentication in the API gateway has the advantage that it hides this complexity from the microservices and relieves them of the responsibility of handling a diverse set of authentication schemes [61]. With the *backends for frontends* pattern, different API gateways can not only provide each consuming application with a custom API but also handle the needed authentication scheme.

The focus of this section is on the message flow between the API gateway and the identity service or STS. To limit the scope of the discussion, the assumption is made that the following three conditions are satisfied in each discussed variation of the message flow. Firstly, the end-user obtains an access token from the identity service and presents the token to the API gateway in each request. Secondly, the API gateway verifies the token and either allows or denies the request to the underlying microservice or microservices. Thirdly, the connection between the end-user and the API gateway is protected with SSL/TLS. Following are some variations of the message flow:

API gateway always invokes the identity service

To verify the user credentials, the API gateway calls the identity service each time before passing the request to the underlying microservice or microservices, as seen from Figure 4.1. The drawback with this approach is that it increases the number of connections to the identity server, and hence reduces the performance and availability of a microservice application. The adverse effects of repeatedly invoking the identity service are especially pronounced if the identity service does not scale on the same level as the API gateway. This may be due to architectural limitations caused by the sensitivity of this piece of infrastructure [67].

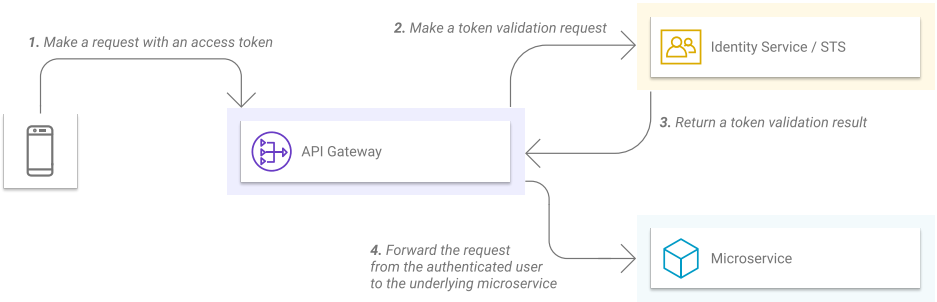

Figure 4.1: Invoking the identity service on each request

API gateway verifies a self-contained token

To mitigate the problem of performance and availability degradation, the API gateway can verify the access token locally, provided that the access token is self-contained and the API gateway can obtain a new certificate of the identity service whenever it is renewed. This removes the necessity of invoking the identity service every time the user presents an access token. The drawback with this approach is that should a self-contained

access token be prematurely revoked at the identity service end, the API gateway would not be aware of this since it no longer interacts with the identity service issuing these access tokens [67].

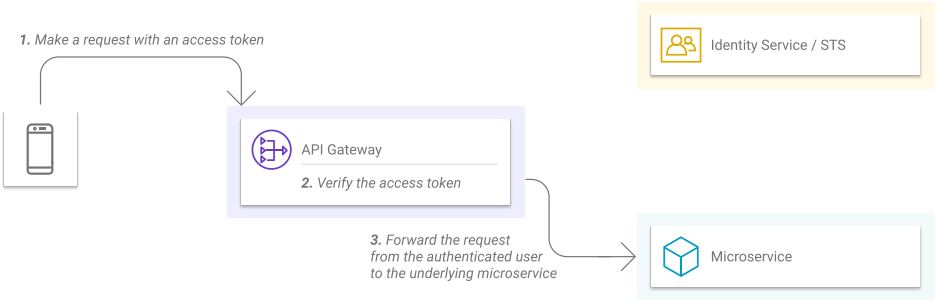

Figure 4.2: Verifying the signature of a self-contained token

API gateway verifies a self-contained token and subscribes to updates in the revocation list

To mitigate the problem of premature access token revocation, the identity service must notify the API gateway of any events of access tokens being prematurely revoked. The API gateway needs to subscribe to updates in the access token revocation list via a Pub/Sub mechanism and maintain a local copy of the revocation list, as seen from Figure 4.3. A subscription can use either the pull or push mechanism. In addition to subscribing for updates of the revocation list, the API gateway must receive a new certificate of the identity service whenever it is renewed.

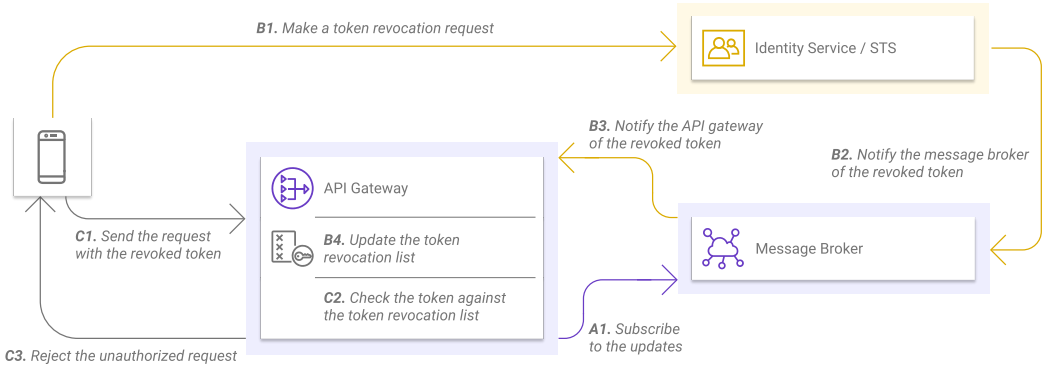

Figure 4.3: Subscribing to revocation list updates

Table 4.1 summarises the above approaches to authentication at the edge regarding the issues they address and required work.

	Always invoke the issuing service	Only verify the signature	Verify the signature, maintain the revocation list
Performance	No	Yes, *drastically* limits the number of remote calls	Yes, limits the number of remote calls
Revocation	Yes, no additional mechanisms are required	No	Yes, requires a Pub/Sub service

Table 4.1: Comparison of approaches to end-user authentication

As seen from the table, the first approach achieves the best security, the second approach wrings the most performance, while the last approach aims to establish a balance between the two. However, it also requires that the additional mechanisms for maintaining an up-to-date revocation list at the API gateway are implemented, which increases the complexity and required development effort.

After the request is authenticated as coming from a specific end-user, the API gateway might want to verify that the end-user is authorised to perform the requested action. Based on the types of AC models presented in Chapter 2, authorisation policy enforcement at the edge can be implemented as follows:

API gateway enforces coarse-grained RBAC policies and more fine-grained AC policies are enforced at the service level

Typically, an ingress API gateway implements coarse-grained RBAC to microservice API paths and methods, while more fine-grained AC to individual domain objects is enforced at the service level; implementing fine-grained AC in the API gateway is complicated since microservices encapsulate their domain logic [61, 67]. To enforce RBAC policies, the API gateway requires information about the identity and roles of the end-user making the request and applicable policies. The API gateway does not store users; instead, it depends on a standalone identity service or similar and simply enforces the policies [13]. The roles assigned to

the end-user can be either included in the request in a cryptographically secure manner, for example, in a JWT, or retrieved from the identity service upon receiving the request. In the former case, the API gateway will not know if the roles have been changed. To mitigate this issue, the identity service can revoke the token issued for this user and notify the API gateway as shown previously in Figure 4.3. In the latter case, the API gateway can cache users and roles to reduce the response time and the number of connections to the identity service.

Implementing RBAC at the edge of the deployment has several limitations. Firstly, it only provides coarse-grained AC, which requires microservices to implement their own authorisation logic. While RBAC policies are considered to be easy to audit by reviewing the roles assigned to the user and enumerating permissions within this set of roles [12, 39], defining and enforcing authorisation policies in different locations risks losing this advantage of RBAC. In addition, if parts of the authorisation logic are defined in source code, changing this logic requires redeploying the microservice. Secondly, having the same set of roles might not satisfy the needs of all microservices, which also creates coupling between microservices and the API gateway or identity service because microservices are dependent on centrally created and assigned roles. Thirdly, enforcing RBAC policies at the edge allows the requests that would be later denied by the underlying microservices to enter the private network. Finally, not all access control policies can be expressed through simple role assignments. For example, a student may be denied access to grades other than their own. This kind of rules must be implemented as a part of microservice domain logic.

API gateway enforces fine-grained ABAC policies, which can be defined at different places

To delegate more complex decision making to an ingress API gateway or proxy, the owner of the microservice must define the policies in a declarative format that would allow the API gateway reach an authorisation decision by evaluating the request and against the policies and data. Policy engines such as OPA and Speedle+ acting as a PDP provide means to evaluate both RBAC and ABAC policies described in a declarative way. ABAC policies offer more flexibility and granularity compared to RBAC policies [12]. This allows microservices to externalise their access control logic.

Based on the case studies in Section 5.2, the general approach to policy distribution is to store them in a centralised location such as an

S3 bucket and notify the PDP such as an API gateway, microservice, or sidecar about policy changes. Policies can be defined per microservice, per deployment, or combine both in a layered approach.

The API gateway can either handle decision making by itself or rely on an independently deployed policy service. Both OPA and Speedle+ policy engines can be embedded inside the API gateway as a library, run as a host-level daemon, or be deployed in a separate Docker container.

API gateway enforces fine-grained CBAC policies based on an unforgeable token

Both RBAC and ABAC policies are identity-based meaning that the PEP requires proof of the user identity before it can allow the request. Besides, the PDP requires the policies to reach an authorisation decision. As opposed to this, an ingress API gateway or proxy can enforce fine-grained CBAC policies based on an unforgeable capability token, such as a macaroon [7]. A capability token already contains all the information needed to make the authorisation decision, and hence no other parties need to be involved in decision making. However, there is no evidence of widespread use of CBAC in web API security and, as discussed in Chapter 2, a traditional authentication mechanism may still be required for user accountability and audit.

Although other approaches and variations do exist, the list above is reasonably inclusive. Regardless of the chosen AC model, implementing authorisation in the API gateway has limitations. Firstly, it inherits the shared issues of having an APA gateway, which may become a single point of failure. Secondly, an ingress API gateway only controls public traffic that enters the microservices deployment, also called north-south traffic, but not the interactions between microservices. Consequently, it does not control private, or east-west, traffic that flows between microservices. If one microservice calls another microservice, the called microservice must handle the authorisation by itself, at the edge of the service. This is discussed in Chapter 5.

4.2 Restricting Access to the Underlying Microservices

Enforcing authentication, authorisation and throttling policies alone in the API gateway does not protect the underlying microservices against requests

coming from outside the microservice deployment. This implies that there must be mechanisms in place to prevent requests originating from outside the microservice deployment from bypassing the API gateway. This section discusses two such mechanisms, firewall rules and mTLS, which can be used individually or in combination.

Restricting access with a firewall

A firewall allows or blocks incoming and outgoing network traffic based on a set of inbound and outbound rules to restrict access and prevent data exfiltration. It can be configured only to allow the API gateway to access the underlying microservices and to reject or drop all traffic that is not expressly permitted by the rules. In an *L3/L4 firewall*, also referred to as a *stateful firewall*, the action is taken solely based on source or destination IP address, port, and protocol. A *L7 firewall*, also referred to as a *context-aware firewall*, can also inspect the content of the packets in addition to everything that the *L3/L4 firewall* do [35].

Restricting access using mTLS

The standard TLS protocol provides both the integrity and confidentiality of the transmitted data, allowing the client to authenticate the server by verifying the presented X.509 certificate signed by a trusted CA and to negotiate an encryption algorithm and cryptographic keys [14]. Mutual TLS (mTLS) refers to the process whereby both parties present their X.509 certificate and prove possession of the corresponding private key. mTLS is a more agile alternative to a firewall. In addition to restricting access, it achieves confidentiality and integrity of the data in transit between the API gateway and a microservice. mTLS verification happens at the L4. As a result, it does not require changes to the application logic of the microservice, which aligns with the single responsibility principle. As an alternative to implementing mTLS in the microservice itself, it can instead be handled by its sidecar proxy and ultimately a service mesh. Restricting the access only to requests coming directly from the API gateway is a special case of service-to-service authentication, which is discussed in more detail in Chapter 5.

Table 4.2 summarises the differences between restricting access with a firewall and mTLS.

	Firewalls	mTLS
Layer	L3/L4 or L7	L4, does not propagate up to the L7
Protocols	Typically TCP and UDP	Supports also messaging protocols and can be used with RabbitMQ, Kafka, and other message brokers to restrict access to reactive microservices
PEP	A firewall	An API gateway, microservice or sidecar proxy
Scalability	Better suited for a static infrastructure, needs to be updated if the IP address of an API Gateway changes	Suited for both static and dynamic environments
Granularity	Better suited for coarse-grained network segmentation	Suited for fine-grained access control
Encryption	No	Yes

Table 4.2: Comparison of firewall rules and mTLS

Chapter 5

Zero Trust Security with a Service Mesh

The traditional perimeter security model breaks the network into zones, each of which is assigned a level of trust. For example, in the edge security architecture similar to the ones presented in Chapter 4, the client exists in the untrusted zone, the API gateway that faces the public internet is placed in the demilitarised zone (DMZ), and the underlying microservices are in the trusted zone, as visualised in Figure 5.1. The perimeter model relies solely on network security, that is, on placing firewalls between zones with different levels of trust, and lacks intra-zone traffic inspection [23].

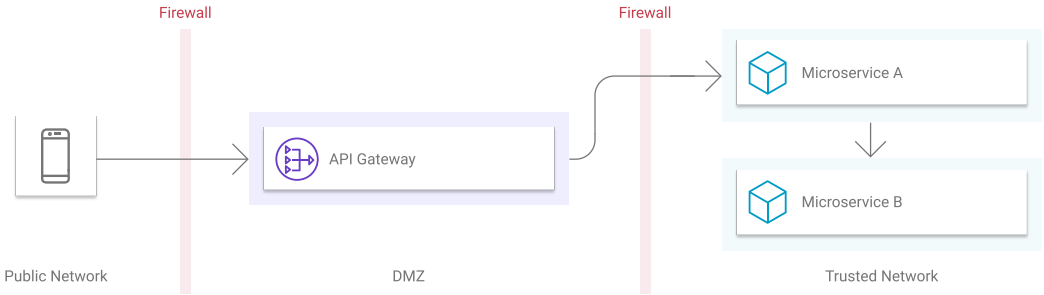

Figure 5.1: Traditional network security architecture

It assumes that each microservice deployed in the private network and the private network itself can be fully trusted, including everything that a microservice claims about itself and the end-user [67]. However, this assumption is flawed and fails to address security threats to a microservice application deployed in a dynamic environment.

A typical attack on a microservice deployment that leverages the perime-

ter model involves a microservice in the "trusted" network consuming a compromised external resource outside the firewall. It allows data to be both infiltrated and exfiltrated from the otherwise-protected network. As this example shows, whilst protecting a microservice deployment at the edge has its benefits, including providing a stronger defence-in-depth by adding an extra line of defence, it is not sufficient [23].

As a result, the perimeter security model is becoming obsolete, and the industry is shifting towards the zero trust security model. In a 2019 Forbes Insights survey[1], 66 percent of more than a thousand security executives and practitioners and 90 percent of companies where security is highly integrated into decision making reported having zero trust networking policies in place [85]. Almost 50 percent of respondents reported a desire to focus their efforts on securing their private networks [75].

As opposed to the perimeter model, the zero trust model assumes that the network is always hostile. Gilman and Barth list five fundamental assertions about a zero trust network [23]:

- The network is never assumed to be trusted.
- The network must be secured both against external and against internal threats, which always exist on the network.
- Neither physical nor logical host placement alone can be used to treat the network as trusted.
- Every actor such as a user or microservice and their every action must be authenticated and authorised.
- Security policies must be calculated dynamically.

In the context of the microservice architecture, this means that each request must be authenticated at every entry point [67], service-to-service interactions must be closely monitored to confine the trust put on individual microservices and to limit the potential damage [72], and firewalls may be used to restrict the potential damage caused by a compromised microservice within the internal network in addition to protecting the internal network from external threats as in the perimeter security model [82].

This chapter discusses steps for incorporating zero trust security into a microservice deployment by securing east-west traffic with a service mesh. These steps include protecting service-to-service communication channels with mutual TLS and JWTs, securely sharing end-user context between

[1]The survey has been conducted as part of the paid program with VMware.

microservices, managing secrets across microservices, as well as additional security measures such as distributed tracing.

5.1 Enforcing Access Control Policies on End-Users at the Edge of the Service

Whereas all the components of a monolithic application share the same web session and the identity of the end-user is retrieved from it, microservices generally do not share resources. In a zero trust microservice application, two conditions must be met. Firstly, a microservice must be able to communicate the identity of the end-user to the services that it invokes. Secondly, a microservices must not trust whatever another microservice claims about the end-user who initiates the message flow between the two microservices. These conditions imply that the end-user must be continuously authenticated at every entry point. To allow authentication, the identity of the end-user can be securely passed from one microservice to another in a self-contained access token such as JWT that carries information about the end-user [61, 67]:

Reusing the same JWT across microservices

It the first approach, a microservice reuses the JWT it received from the upstream microservice and passes it to the downstream microservice, as shown in Figure 5.2. Since the audience of the JWT token remains the same, all or at least the majority of the microservices in the deployment must accept the same audience value.

Requesting a new JWT for each service interaction

In the second approach, a microservice requests a new JWT from the STS in exchange for the JWT it receives from the upstream microservice, as shown in Figure 5.3. The STS is either the identity service or a service that trusts the identity service which issued the original access token. The STS issues a new JWT in exchange for the old JWT under a new audience and signs it. It can also populate the "jti" claim with a unique identifier for the JWT to prevent the JWT from being replayed [33]. The token exchange approach allows the STS to deny the request if the calling service must be denied access to the called service based on the authorisation policies. However, service-to-service authorisation is typically implemented at the service level, as discussed in Section 5.2.

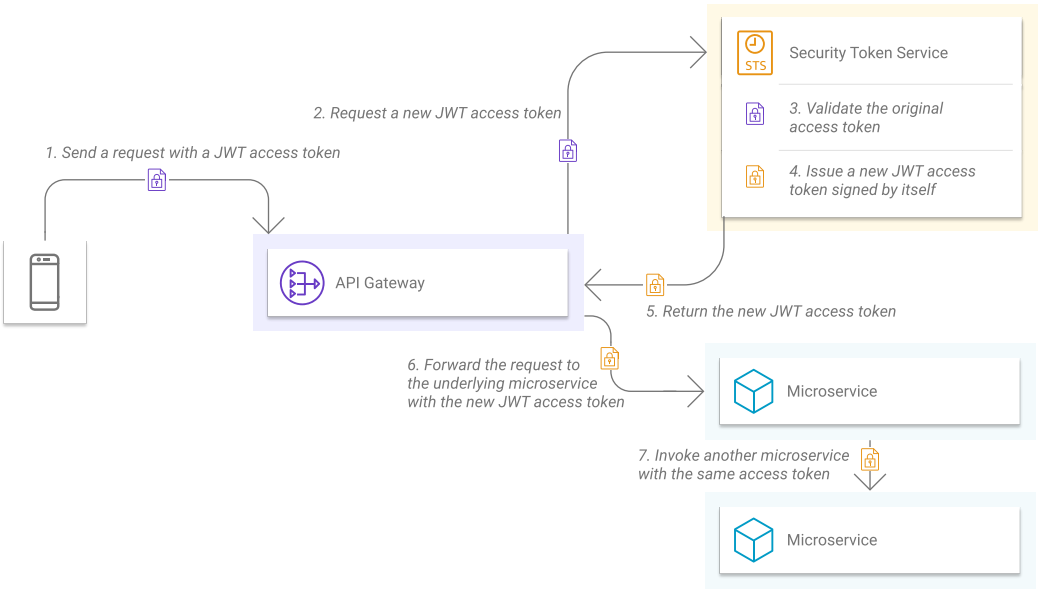

Figure 5.2: Reusing the access token with the same audience value to authenticate the user at each entry point

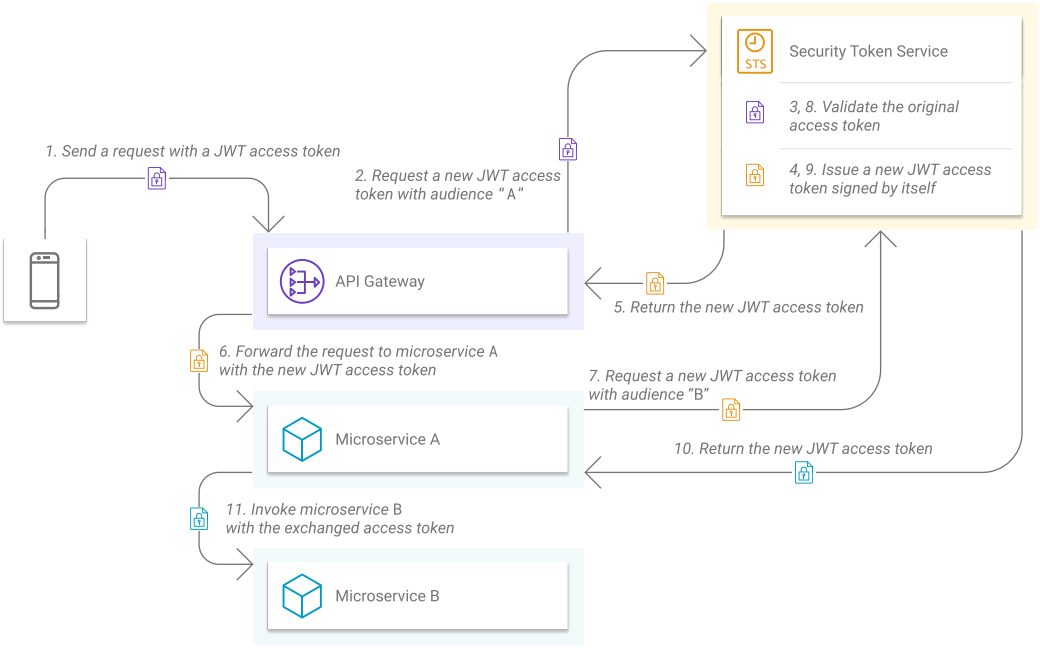

Figure 5.3: Requesting a new access token with a new audience value for each service interaction to authenticate the user at each entry point

A microservice can either implement end-user authorisation by itself like a monolithic application or offload it to a sidecar to adhere to the single responsibility principle. Delegating authorisation to a sidecar inherits the benefits and drawbacks of the sidecar pattern discussed in Chapter 2. Since a microservice and its sidecar run in the same pod, neither a firewall nor mTLS between the two is required.

Suppose all of the services in a heterogeneous microservice application reuse the same sidecar implementation. All the sidecars can form the data plane of a service mesh. As a best practice for operating containers, the container image has to be immutable. As a result, authorisation policies must be dynamically updated in memory via the push or pull model. The control plane of the service mesh can provision up-to-date authorisation policies to the sidecars. For example, the Istio service mesh extracts the identity of the end-user from a JWT and allows specifying authorisation policies based on that. The authorisation policies are stored in the config store and distributed by Istiod to each sidecar, along with the keys where appropriate [30].

The authorisation sidecar can act as the PDP or both as the PDP and PEP. In the former case, microservices can also share a library or package to simplify and unify interaction with the authorisation sidecar [37]. In the latter case, the authorisation sidecar must be a sidecar proxy and accept all the incoming requests before they can reach the microservice.

Envoy sidecar proxy is commonly used for end-user authentication as it supports JWT end-user authentication out of the box [16], while policy engines such as OPA deployed as a sidecar can handle authorisation policy decision making [54].

The benefits and drawbacks of implementing different access control models such as RBAC, ABAC, and CBAC in a sidecar are much the same as when these are implemented in an ingress API gateway or proxy, as discussed in Chapters 2 and 4.

5.2 Enforcing Access Control Policies on Peer Services at the Edge of the Service

In a typical microservice deployment, microservices interact with one another using IPC mechanisms such as HTTP-based REST, WebSockets, and gRPC, message-based AMQP and STOMP, or binary TCP and UDP. Regardless of the IPC mechanism, inter-service communication channels must be protected against eavesdropping and tampering. Microservices must have a way to authenticate one another to comply with zero trust networking

principles [67, 82]. East-west traffic can be secured, inter alia, by using a secure channel (e.g., mTLS with X.509 certificates) or public-key encryption and signature (e.g., JSON Web Signature (JWS) and JSON Web Encryption (JWE)):

Service-to-service authentication with mTLS

mTLS enables microservices to mutually authenticate one another and protects the integrity and confidentiality of the data in transit. However, it does not provide non-repudiation.

To enable mTLS, each microservice must hold a valid X.509 certificate signed by a certificate authority (CA), trusted by all the microservices. If none of the microservice entry points is exposed outside the internal network, but they are instead hidden behind a reverse proxy, a private CA is sufficient. A cloud provider may offer services for creating a private CA. For example, Amazon offers AWS Certificate Manager (ACM) Private Certificate Authority (CA). Alternatively, OpenSSL can be used to set up a private CA.

mTLS is a standard method for implementing service-to-service authentication. Many service mesh implementations such as Istio, Kuma, Aspen Mesh and Linkerd provide mTLS as an out-of-the-box functionality [30, 40, 43]. Istio automatically configures service sidecars to use mTLS when calling other services [30]. However, while suitable for synchronous communication, mTLS cannot be used to protect asynchronous communication and storage of protected messages.

Service-to-service authentication with JWS

JWS represents content secured with digital signatures or MACs using JSON-based data structures. A JWS object consists of three concatenated Base64-encoded segments separated by periods: the JOSE header, JWS payload and JWS signature [32].

Digital signatures have two advantages over mTLS. Firstly, they facilitate non-repudiation. Secondly, they enable asynchronous communication and storage of protected messages.

To protect the confidentiality of the data in transit and at rest, the content of the messages must also be encrypted. JWE represents encrypted content using JSON-based data structures. In the JWE compact serialisation, a JWE object consists of five concatenated Base64-encoded segments separated by periods: the JWE protected header, JWE en-

crypted key, JWE initialisation vector, JWE ciphertext and JWE authenticated tag [34].

While JWS and JWE can be nested in any order, digitally signing a message and then encrypting the signed message should be preferred since this prevents the signature from being stripped from the message and provides privacy for the signer [33].

Table 5.1 summarises how secure channels and public-key cryptography compare to each other. Regardless of whether east-west traffic is secured using mTLS or JWS and JWE, X.509 certificates and private-public key pairs must be stored and distributed across a dynamic environment.

Security goals	mTLS	JWS
Service-to-service authentication	Yes	Yes
Confidentiality	Yes	Yes, if paired with JWE
Integrity	Yes	Yes
Non-repudiation	No	Yes
Asynchronous communication	No	Yes

Table 5.1: Comparing establishing trust with mTLS and JWS

After the two microservices mutually authenticated one another, the called service should verify that the calling service is allowed to perform the requested action.

While ACLs are not well suited for defining and enforcing AC policies in a microservice application with many end-users, they are fit for defining and enforcing AC policies on peer microservices:

- Each microservice is typically owned by a team that also manages the permissions to access the microservice. With the microservice being a resource and the team being a resource owner, this corresponds to discretionary access control (DAC). By defining an ACL for their microservice, the team can maintain complete control over which other services can access the microservice and forbid delegation of these access rights.

- The number of services in a microservice deployment is typically small in relation to the number of end-users.

Based on the reviewed approaches to addressing peer authorisation employed by businesses that build their services on top of the microservice architecture, service-to-service authorisation solutions are primarily ACL-based:

Monzo

Monzo is an online bank based in the United Kingdom. They adopted the microservice architecture from the start in February 2015 and had over 1,500 microservices with more than 9,300 unique connection as of the end of 2019. As a bank, they work towards a completely zero trust platform [36]. Each service has a manually approved list of allowed services, which has, on average, six items. The policies are enforced using the Kubernetes NetworkPolicy API. However, due to a large number of services and connections, maintaining the lists without automatisation would be impracticable. As a result, Monzo wrote a tool called `rpcmap`, which is triggered whenever new code is pushed to GitHub. `rpcmap` scans code for requests to other services and generates a rule file per called service. The name of the file `service.calling/egress/service.called.rule` represents both the calling service and the called service. GitHub then asks a team building the called service to review the rule file `*/service.called.rule` and allow the calling service as ingress source by adding its label to a `NetworkPolicy` object. With this approach, Monzo aimed at minimising the number of manual steps that are prone to errors and human factors.

LinkedIn

LinkedIn is a social network geared to businesses and professionals which run more than 700 microservices as of 2019. Each service defines a list of services and permissions. The authorisation check is performed by an authorisation client module that runs on each service. To reduce latency, the service keeps a copy of the ACL in-memory and makes an authorisation decision locally. The original ACL is stored in an Espresso database fronted by an external Couchbase cache. A change data capture system based on Brooklin notifies the service if the original ACL is changed. ACLs are managed through a cloud management interface Nuage or a command-line tool. However, LinkedIn does not reveal the process of defining the ACLs [42].

Dropbox

Dropbox is a cloud storage service which runs hundreds of Courier services. Courier is a gRPC framework for service development created by Dropbox. Each service holds a TLS certificate issued by a private CA. Services mutually authenticate each other with TLS 1.2+. After the identity of the calling service is confirmed and the request is decrypted, the called service verifies that the calling service has proper permissions. ACLs can be set on both services and individual methods. Courier handles subscribing for policy updates and fetching new versions as they become available, relieving the development team from this responsibility [52].

Other microservice-driven businesses approach service-to-service authorisation by enforcing ABAC policies:

Atlassian

Atlassian develops cloud products for software development and project management, which include JIRA, Trello, Bitbucket, and Confluence. These products rely on thousands of heterogeneous microservices with tens of thousands instances that run in AWS, their in-house Kubernetes cluster, and other locations.

Atlassian implements the perimeter security model by partitioning the network into the customer, DMZ, and internal zones. Each zone entry point is protected by an API gateway that enforces security policies on inter-zone traffic. Both API gateways and services utilise an internal HTTP request authentication tool called SLAuth, which stands for Service-Level Authentication. In addition to authentication, SLAuth implements authorisation. To support authorisation, it embeds OPA. SLAuth has several deployment models and can be embedded in an application or Envoy sidecar proxy.

Policies are written in a JSON-based format and translated into Rego. Before policies reach SLAuth, they are submitted to a policy register, which validates, tags, and stores them in an S3 bucket. The policies are searched via environment, region, and service tags and served via a content delivery network (CDN).

Atlassian distinguishes between platform and service policies. Platform policies are enforced across the entire platform or a segment of a platform to provide a basic level of security to all services, while service policies are written by service owners and only apply a single service.

This model is used to authorise peer services and CI/CD systems. The process of defining policies is mostly manual [71].

Pinterest

Pinterest is a visual discovery engine which runs thousands of instances of hundreds of microservices.

With some exceptions, each microservice is deployed alongside an Envoy sidecar proxy and another sidecar that wraps around the OPA engine. Envoy authenticates peer services and CI/CD systems by using mTLS. X.509 service and system certificates include information about the identity of their owner. When Envoy receives an API request destined for the microservice, it passes the identity of the calling service or system to the OPA sidecar that decides if the request should be allowed or denied. In addition to decision making, the OPA sidecar registers with ZooKeeper to receive notifications about policy changes and fetches policies from an S3 bucket if needed.

To enforce an OPA policy, the developer must submit a corresponding Rego file in a pull request. The pull request must be reviewed and approved by another member of the development team or a member of the security team. The approved pull request triggers a build on an isolated build system, which uploads the policy to the S3 bucket and triggers ZooKeeper.

This model of policy distribution and decision making also applies to end-user authorisation [37].

The way of enforcing AC is usually uniform across the whole microservices application. The mentioned ways include relying on Kubernetes Policies, using custom libraries developed by a dedicated security team and using a service mesh of sidecars.

Most of the reviewed company tech blogs or white papers have no mention of how they compile the ACL and what are the human processes behind creating and approving ACLs except for Monzo, which claims to rely heavily on automation.

Chapter 6

Migrating to Microservices

This chapter describes the process of migrating from a monolithic to a secure microservice architecture. The existing monolithic application is an online learning platform used in *CS-C3130 Information Security* course organised by the Aalto University School of Science. For simplicity, it is referred to as *the application* throughout this and the following chapters.

According to the survey of 21 microservice adopters, practitioners adopt microservices to encourage team autonomy and ownership, reduce time to market, scale up and down cost-effectively, increase fault tolerance and catastrophe preparedness, improve maintainability, enable DevOps, experiment with new technologies and adhere to the mainstream [74]. The reasons for migrating the monolithic online learning platform to a small set of microservices include solving the existing maintainability issues, improving the readiness of the application for DevOps, and experimenting with new technologies and ways of building secure microservice applications.

This chapter examines the architecture and functionality of the original monolithic application, lists the requirements that need to be taken into consideration during the migration process, describes the migration process, and evaluates the resulting microservice architecture.

6.1 The Starting Point for the Migration

The application is a monolithic Model-View-Controller (MVC) application written in Python using the Flask framework. Its primary purpose is to provide students with a user interface for launching the exercises and submitting a solution for each part of the exercise. Hence, it is referred to as the *launcher*. The launcher also provides visualised statistics on the progress of the students in the class.

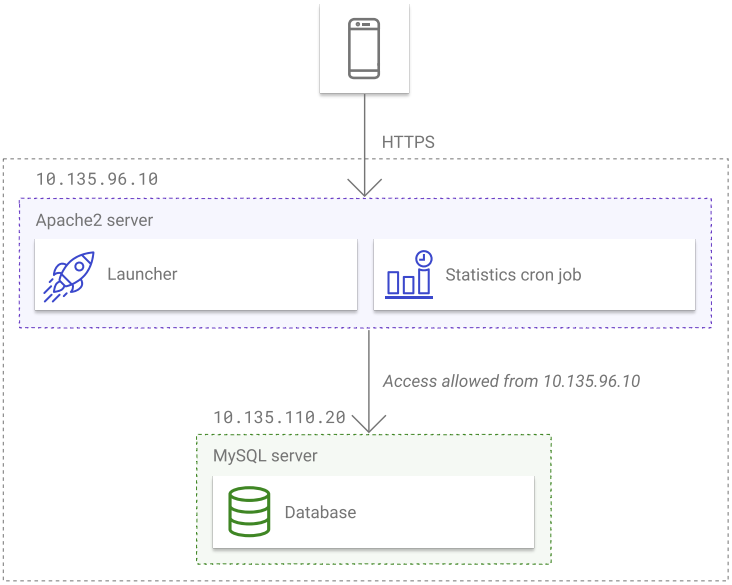

Figure 6.1: The architecture of the monolithic application

The launcher is deployed in the Apache web server and secured with HTTPS with a certificate from Let's Encrypt, which is a free, automated, and open CA. It uses a MySQL database to store user data. Both the launcher and the database are assigned static IP addresses in a private IP address space. The database allows connections only from the IP address of the launcher.

The database has two tables, `users` and `marks`. The `users` table contains the following fields: `id`, `first_name`, `last_name`, `email`, `student_number`, `user_number`, `password`, and `salt`. The `marks` table contains the following fields: `id`, `user_id`, `exercise_id`, `problem_id`, `points`, and `state`. The `user_id` field in the `marks` table is a foreign key that points to the primary key of the `users` table. Flask handles access to the data and database migrations via object-relational mapping (ORM).

Accounts

The launcher is not integrated with Aalto Shibboleth single sign-on (SSO) system for online services. As a result, the student needs to go through a separate process of authentication and authorisation. The student creates an account by submitting a form with their full name, email address, student number, and password.

The launcher does not currently validate or use the provided email, nor can it be used for a password reset. A password reset can only be performed

by a member of the course staff.

The launcher does not impose restrictions on the password. The password is hashed, and the hash and salt are stored in the database. A secret used for hashing the password is stored in a Git repository alongside the source code.

The launcher generates a random user number, which is used in the rest of the system in place of the other personal identifiers for better GDPR compliance.

The student authenticates with their email and password. The information specific to the student is stored from one request to the next in a session, which is implemented on top of cryptographically signed cookies. The secret key used for signing is also stored in the Git repository. These tasks are handled by `flask_login`.

Exercises

The configuration settings for the exercises and problems are defined in a JSON configuration file, which must be edited to open or close an exercise or a problem within an exercise. The launcher starts exercises within their own Docker containers upon student request. It manages containers using a Python library for the Docker Engine API. The launcher specifies a name for each container, which is used when referencing the container within a Docker network.

The random user number is provided to the container as an environment variable and used to personalise the excesses. Access to a running Docker container is not restricted to the student who launched the exercise. However, the applications running in the Docker containers are typically password and do not contain personal information.

The launcher evaluates the problems automatically and stores the grades in the database. The database schema can also be seen in Figure 6.1.

Statistics

A scheduled job produces the statistics and saves them to a text file that can be accessed by the launcher and an authorised member of the course staff. The launcher anonymises and visualises the data, while an authorised person can access the produced text file that also contains identifying information of the students.

6.2 Considerations Before the Migration

The requirements that need to be taken into consideration can be classified into the following categories: trust and security, performance and costs, deployability, and maintainability. These categories are elaborated upon in this section.

Trust and Security

A few assets were identified that might be of interest and must be protected:

- *Confidentiality and integrity of personal data.* The application stores personal data of students. There is the identifying information (first name, last name, email, and student number) and the exercise results and log files. This personal data must be protected following the rules of the GDPR.
- *Integrity of the grades for the exercises.* The application evaluates the exercises and stores the grades in the database. These grades must be protected from tampering.
- *Prevention of accidental loss, destruction, or damage of identifying information and the grades for the exercises.* The final grade for the course depends on the results of the online exercises. Therefore, they must not be lost, destructed, or damaged.
- *Confidentiality of the solutions to the exercises.* The solutions must not be known to students before solving the problem.
- *Confidentiality of the source code of the exercises.* The exercises must remain a black box for students.
- *Availability.* The application should be available at all times, especially close to the exercise submission deadline.

Performance and Cost

As evident from the previous chapters, the challenges of securing microservices do not solely concern the security aspects of a microservice application but also other aspects such as performance and cost. The best security cannot be achieved with the most performance for the least cost. For that reason, a prioritisation of the goals may be needed while planning a migration [50]. In the discussed case, performance is not critical since the application is neither

a latency-critical application nor a commercial product. Before the exercises were containerised, there was an issue of high memory usage; however, this issue no longer exists.

Deployability

The deployment process is not automated and involves manual steps, described in the `README` file. As a result, it is time-consuming and error-prone. Due to the deployment complexity of the microservice architecture, DevOps is a prerequisite to adopting the microservice architecture.

Maintainability

The application is typically maintained by a small number of course assistants that rotate frequently.

Taibi et al. [74]	Richardson [61]	Newman [50]	Palladino [56]
Gradually reifying the existing features into services			
Yes, with two variations	Yes, mentioned as the *strangler fig* pattern	Yes, mentioned as the *strangler fig* pattern, strongly recommended	Yes, mentioned as the *ice cream scoop* strategy
Separating the presentation tier and backend			
-	Yes, possibly a variation of the above strategy	-	-
Only building new features as services			
Yes	Yes	-	Yes, mentioned as the *Lego* strategy
Rewriting the whole monolith into microservices all at once			
-	Yes, mentioned as the *Big Bang rewrite* strategy, not recommended	Yes, not recommended	Yes, mentioned as the *nuclear option* strategy, acknowledged to be rarely adopted

Table 6.1: Strategies for refactoring a monolith to microservices

6.3 Migration Process

Table 6.1 summarises the four strategies for migrating from a monolith to microservices found in the literature. Out of the four strategies, the gradual refinement and the Big Bang rewrite strategies are the ones that can be adopted in the current work. However, the Big Bang rewrite strategy is rarely adopted and generally not recommended since it is likely to stall the development of the features and slow down the response time to new market conditions and user needs, and therefore lacks one of the main advantages of the microservice architecture, which is agility and faster time-to-market. Taking this into account, this work adheres to the gradual refinement strategy. Both Richardson [61] and Newman [50] refer to this application modernisation strategy as the *strangler fig application* pattern, a term introduced by Fowler [20]. This section describes the process of decoupling a service from the monolith as the first stage of the migration. The process roughly follows the steps identified by Newman [50].

6.3.1 Decoupling a Service from the Monolith

A migration should typically start by decoupling edge capabilities, as opposed to capabilities that are deeply embedded in the monolithic application [21, 50]. This makes the identity service a good candidate for the first microservice.

Priority	Requirements
1	• It must provide functionality for creating an account that has at least the first name, last name, email address, and student number of a student. • It must generate a random user number for each student. • It must issue an access token in exchange for a valid email and password. • The private key used for signing access tokens must be kept secret.
2	• It should issue short-lived access tokens. • It should provide a token revocation endpoint.
3	• It should log all the requests. • It should provide the API reference documentation. • It should expose a JWKS endpoint.

Table 6.2: Requirements for the new identity microservice

Specifying the requirements for the new microservice

Refactoring a monolith to microservices should be a behaviour-preserving change [49]. This also affects the requirements for the new microservice. Table 6.2 lists the requirements in the order of priority, where 1 denotes the highest priority and larger integers denote lower priorities. Table 6.3 lists the API methods of the future microservice that met the requirements.

Method	Description
`POST /users`	Creates a user.
`POST /tokens`	Validates the provided credentials and issues a short-lived JWT access token.
`GET /tokens/validity`	Validates the JWT access token.
`POST /tokens/revoke`	Revokes the JWT access token.
`GET /.well-known/jwks.json`	Exposes a JWKS endpoint.

Table 6.3: Identity service API methods

Embracing the polyglot nature of the microservice architecture

The code of the future microservice can be either extracted from the monolith or rewritten [50]. The amount of code implementing the domain logic for generating user numbers, hashing passwords and validating credentials is small, especially when compared to the amount of boilerplate code that handles starting a server, connecting to a database, request routing, and error handling. As a result, we decided to rewrite and retire the old code. This also allowed us to embrace the polyglot nature of the microservice architecture and use TypeScript with Node.js and Express.

TypeScript is a gradually-typed superset of JavaScript that adds static type-checking via type annotations. Enforcing the use of type annotations by configuring `typescript-eslint` to disallow usage of the `any` type eases bug detection. Based on the results of the empirical study by Gao et al., TypeScript could have prevented 15 percent of the bugs in large, mature projects publically available on GitHub [22].

While the old code contained mutable variables and threw exceptions, the new code is immutable and handles errors in a functional fashion.

The new identity microservice uses TypeORM to access the database which supports the original MySQL database. The benefits of using an ORM include the lack of direct manipulation with SQL, which protects from SQL

injections. The identity microservice also relies on Redis to store the access token revocation list.

The identity microservice, Redis and MySQL database are deployed as Docker containers. The Docker environment is described and built with Docker Compose using the Infrastructure as Code (IaC) approach. In addition to service definitions, the Compose file contains network and volume definitions. The network is configured with static IP addresses. Each container is reachable by other containers on that network.

Testing the newly formed microservice in isolation

Before the identity service is integrated into the application, it is tested in isolation with Jest.

6.3.2 Putting the Pieces Together

After the new identity service is tested in isolation, it can be integrated into the launcher:

Configuring the old services with Docker Compose

Firstly, a Dockerfile is created for both the launcher and the scheduled job that produces statistics. The definitions for these services are added to the Compose file.

Integrating the newly formed microservice into the monolith

Secondly, existing calls to the methods of class `User` are redirected to the new service. This step also required additional changes in the codebase which were not directly related to redirecting the calls but rather to the transition from session-based to token-based authentication. While the new identity service provides endpoints for token validation and revocation, the launcher ignores these and instead verifies the validity of the token locally. When the user logs in with their email and password, the launcher requests a new JWT from the identity service in exchange for these credentials and saves the token in a cookie.

Removing the not needed parts from the monolith

Thirdly, the no longer needed code is deleted from the launcher codebase. This includes, for example, the whole `User` model.

Setting up HTTPS

Finally, the definition for a NGINX reverse proxy is added to the Compose file. NGINX is configured to use a Let's Encrypt certificate and expose the port 443 for inbound HTTPS traffic. It passes the client request to the launcher, fetches the response, and sends it back to the client.

6.3.3 Decomposing the Database

Loose coupling and high cohesion are one of the key characteristic features of microservices. To satisfy these constraints imposed by the microservice architecture, each microservice has to keep the data private in a separate database [49, 61]. At the same time, Taibi et al. identify database migration and data splitting as the second most reported migration issue [74].

While data migration on a live application poses a challenge, this is not a concern in the present case since the application is publicly available only during the periods when the course is running.

With only the identity service extracted from the monolith, the resulting architecture would have to follow the shared database pattern because of the statistics job which uses data from both the `users` and `marks` database tables. However, sharing a single database among two or more services causes a couple of issues. Firstly, with a shared database, there is no clear separation as to which service owns the database schema or understanding about what parts of the schema can be changed safely, causing no effect on the functioning of other services. Secondly, allowing all services access to any data violates the principle of least privilege. The launcher would have unnecessary access to sensitive user data, which includes student numbers and password hashes and salts.

The chosen approach to mitigate the above issues was to split the database to have one database per service and to transform the statistics job into its own microservice. The statistics microservice is allowed read access to both databases. This does not fully resolve the issue of coordinating the changes between the microservices; however, it does restrict the launcher from accessing sensitive user data stored in the database.

6.4 Evaluating the Naïve Microservice Architecture

The steps described in the previous section result in the final strangler fig architecture, as shown in Figure 6.2.

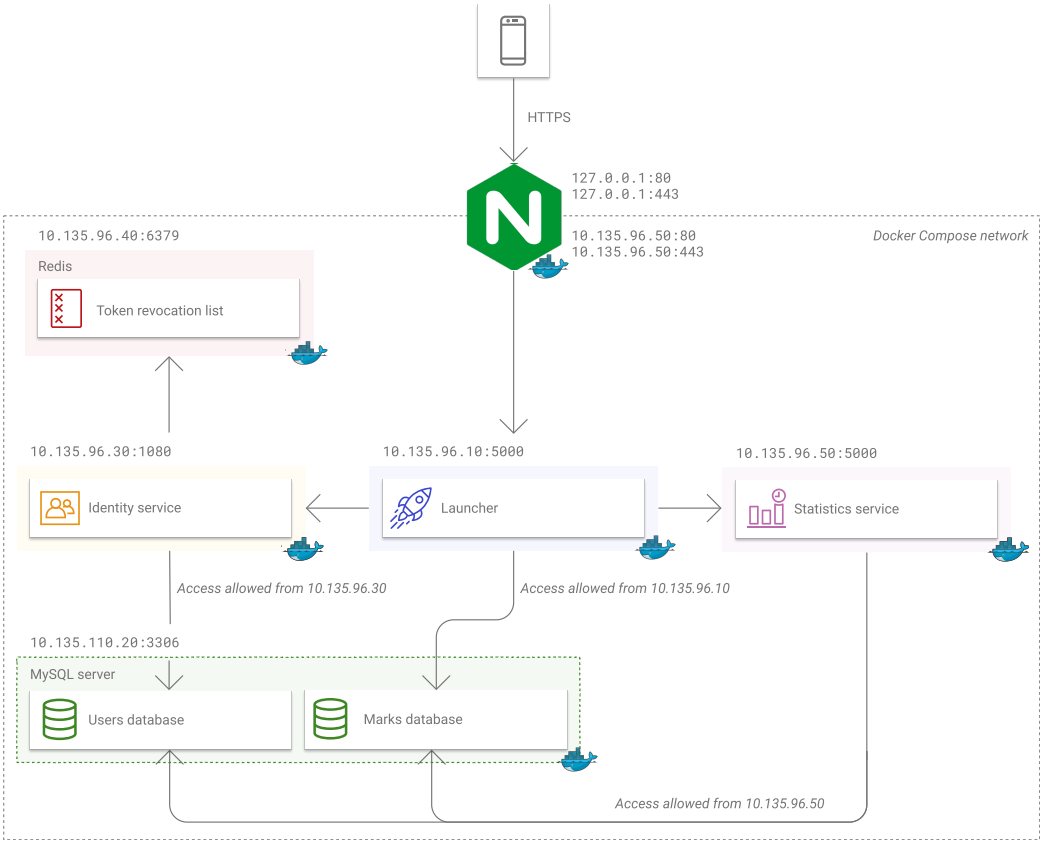

Figure 6.2: The naïve architecture of the microservice application

The resulting architecture is referred to as "naïve" since it was implemented prior to reviewing the literature summarised in Chapters 4 and 5. Below are the achievements and shortcomings of the migration evaluated based on the knowledge summarised in the previous chapters of the thesis:

Deployment automation

The adoption of the IoC approach with Docker Compose has simplified the deployment process by reducing the manual overhead of installing dependencies and configuring service dependencies. However, the static

IP address assignment makes the configuration relatively static, which reduces the agility of microservices. Dynamic IP address assignment, on the other hand, would allow microservices to start and terminate quickly to scale up and down. However, it would also present a new challenge of service discovery.

Separation of concerns

The User domain model that is responsible for implementing any business rules related to users is decoupled from the rest of the application logic, which also helps to isolate the more security-critical parts of the application. Nevertheless, the launcher remains partially responsible for authentication and authorisation concerns. It verifies the JWT upon every request to a protected API endpoint and retrieves the identity of the user from the JWT (Appendix A). The launcher also verifies if the user is permitted to access the requested resource or perform the requested action. For example, it asserts that the problem is open before evaluating the solution to the problem. Instead of implementing these checks in the launcher code, authorisation decision making could have been delegated to a dedicated sidecar. Assuming that the routes are changed to contain also the unique identifier of the user `uid` and the token is passed in a more conventional manner in the `Authorisation` header, the same authorisation logic can be expressed in Rego.

Secrets management

The secrets are provided as environment variables. They are no longer stored in the Git repository, nor they are embedded in the Docker image. As a result, if a secret has to be added or revoked, the image does not need to be rebuilt. This aligns with the best practice of keeping containers immutable, as discussed in Section 3.5.

Database per service

With two separate databases, the launcher no longer has access to sensitive user data at rest, which aligns with the principle of least privilege. However, the statistic microservice still has access to information such as password hashes and salts that it does not need.

Trust between microservices

Overall, the implemented architecture corresponds to the perimeter security model. Only the NGINX proxy is exposed outside the Docker Compose deployment, while all other services communicate within the private network and only the launcher is semi-exposed to the public internet via the NGINX proxy. Within the private network, the services trust one another and the network which contradicts the zero trust networking principles. The east-west traffic is not protected.

Other concerns

The resulting architecture does not fully address the existing issues such as lack of integration with Aalto SSO and lack of access control to exercise containers. The issue of lack of password reset is addressed partially. Password reset can now be performed by requesting a reset link from the course staff. If deployed on a cloud platform that allows outbound SMTP, the same reset link could be emailed to the student.

Chapter 7

Conclusion

The microservice architecture is a still-young architectural style, which is rapidly gaining widespread adoption. In the 2020 O'Reilly survey on microservices adoption, three-quarters of 1502 respondents reported their organisations adopting microservices, a majority of which started using microservices in the past three years. At the same time, only slightly more than 10 percent of organisations adopted microservices more than five years ago [44]. The microservice architecture is driven by the industry, which results in a lack of academic research and a wealth of grey literature such as company blogs, white papers, and conference talks. Grey literature provides a valuable resource for understanding the microservices architecture and gaining insight into current practices. This thesis references a number of such resources, such as Monzo, LinkedIn, and Dropbox engineering blogs and conference talks given by engineers from Atlassian and Pinterest.

Besides being a buzzword, the microservice architecture tackles real problems of the monolithic architecture, which are discussed in Section 2.1. However, the adoption of the microservice architecture also affects how security must be approached. Chapter 3 lists the novel architectural and organisational challenges of microservices, which include protecting a larger attack surface; striking a balance between security and performance; establishing trust between microservices; sharing user context; managing policies and secrets; collecting logs, metrics, and distributed traces; and bringing diverse security expertise to secure heterogeneous microservices.

Both the edge of the microservice application and the communication between microservices within the application need to be secured: while securing the application at the edge reduces the attack surface, this should not lead us to downplay the importance of securing each microservice at the service level. This need is consistent with the tendency towards adopting the zero

trust security model, which is evident in industry [75, 85]. Protecting the microservice deployment at the edge of the deployment and at the edge of the service are discussed in Chapters 4 and 5 accordingly.

Chapter 4 focuses on such aspects of edge security as authentication and authorisation of end-users at the edge of the deployment and preventing requests from bypassing these security checks, which are typically handled by an API gateway or a dedicated edge service. Implementing authentication and authorisation at the API gateway level requires a trade-off among security, performance and complexity. After the request is authenticated as coming from a specific end-user, it needs to be authorised. On a high level, there exist three approaches to handling end-user authorisation in the microservice architecture: performing all the checks at the edge; performing coarse-grained authorisation at the edge and fine-grained authentication at the service level; and performing all the checks at the service level. Since implementing fine-grained security checks solely at the edge of the deployment is complicated and insufficient for zero trust security, microservices typically take the dual approach. The two commonly applied authorisation patterns suggest either enforcing coarse-grained RBAC policies at the edge and more fine-grained AC policies at the service level or enforcing ABAC policies both at the edge and at the service level. The latter approach is adopted, for example, by Atlassian and Pinterest. Implementing ABAC is made more straightforward with emerging policy engines such as OPA. The use of such engines also fosters better separation of concerns since the decision making can be offloaded to a separate service. Access to the microservices behind the API gateway is typically restricted using mTLS, firewalls, or a combination of both.

After the API gateway reaches the positive authorisation decision, the information about the end-user needs to be sent to the underlying microservices. To adhere to the principles of zero trust security, this must be done in a cryptographically secure manner. In addition, each microservice must be able to pass the identity of the end-user to a microservice it invokes, which in turn must be able to verify both the claims about the identity of the end-user and the identity of the calling microservice. In other words, securing a microservice at the service level requires reliably verifying the identities of both the end-user and the upstream microservice.

Chapter 5 focuses on such aspects of zero trust security as authentication and authorisation of end-users and peer-services at the service-level. Authentication and authorisation can be implemented within the service itself or offloaded to a sidecar, which, together with other sidecars, can form the data plane of a service mesh. The latter allows for separation of concerns, reuse, centralised policy management and scalable policy distribution at the

platform layer. Whilst other options exist, JWT and mTLS are de facto industry standards for end-user and service-to-service authentication accordingly. When offloaded to a sidecar, end-user authorisation often relies on a policy engine such as OPA. The same sidecar can also handle peer service authorisation. Alternatively, service-to-service AC policies can be defined in the form of an ACL, which allows for extremely fine-grained AC. The use of ACLs is possible due to a relatively small number of microservices in the deployment. However, even if the number of services does not exceed a thousand, manually whitelisting microservices poses a cumbersome and error-prone task. Thus, it requires a high level of automation. Monzo describes in great detail how they make this process nearly fully automated. However, the same approach would not be possible if their microservices were built using different technological stacks. As future work, different approaches for compiling the ACL for peer services in a heterogeneous microservice architecture can be investigated.

Finally, Chapter 6 describes a case study where we took the first steps of migrating a monolith to microservices, which included extracting the identity service from the monolith. The resulting architecture might not be considered a microservice application but rather a strangler fig application, since most of its functionality remains in the monolith. The resulting architecture is evaluated based on the knowledge accumulated in Chapters 4 to 5 of the thesis. The main achievement of this first step is the improved readiness of the application for DevOps.

Bibliography

[1] *8 Surprising Facts about Real Docker Adoption*. Datadog. June 2018. URL: https://www.datadoghq.com/docker-adoption/.

[2] D. An. *Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed*. Google. Feb. 2018. URL: https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/.

[3] R. Anderson and S. Smith. *ASP.NET Core Middleware*. Microsoft. July 2020. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/.

[4] *Application Deployment and Testing Strategies*. Google. URL: https://cloud.google.com/solutions/application-deployment-and-testing-strategies (visited on 05/24/2020).

[5] G.K. Aroraa, L. Kale, and K. Manish. *Building Microservices with .NET Core*. Packt Publishing, 2017. ISBN: 9781785887833.

[6] *Best Practices for Operating Containers*. Google. Apr. 2020. URL: https://www.datadoghq.com/docker-adoption/.

[7] A. Birgisson et al. "Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud". In: *Network and Distributed System Security Symposium*. 2014.

[8] B. Burns. *Designing Distributed Systems*. O'Reilly Media, Feb. 2018. ISBN: 9781491983645.

[9] R. Chandramouli. "Security Strategies for Microservices-based Application Systems". In: *Special Publication 800-204*. Gaithersburg, MD: U.S. Department of Commerce, National Institute of Standards and Technology, Aug. 2019.

[10] M. Clark. *How the BBC Builds Websites That Scale*. Jan. 2018. URL: https://www.creativebloq.com/features/how-the-bbc-builds-websites-that-scale/.

[11] *Consul Connect Documentation*. URL: `https://www.consul.io/docs/connect` (visited on 10/10/2020).

[12] E. Coyne and T. R. Weil. "ABAC and RBAC: Scalable, Flexible, and Auditable Access Management". In: *IT Professional* 15.3 (2013), pp. 14–16.

[13] C. Davis. *Cloud Native Patterns*. O'Reilly Media, May 2019. ISBN: 9781617294297.

[14] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Internet Engineering Task Force (IETF), Aug. 2008. URL: `https://tools.ietf.org/html/rfc5246`.

[15] N. Dragoni et al. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. Ed. by M. Mazzara and B. Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 9783319674254. DOI: `10.1007/978-3-319-67425-4_12`. URL: `https://doi.org/10.1007/978-3-319-67425-4_12`.

[16] *Envoy Documentation v1.14.1*. URL: `https://www.envoyproxy.io/docs/envoy/latest/`.

[17] *eXtensible Access Control Markup Language (XACML) Version 3.0*. OASIS Standard. OASIS, Jan. 2013. URL: `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`.

[18] D. F. Ferraiolo, D. Gilbert, and N. Lynch. "An Examination of Federal and Commercial Access Control Policy Needs". In: *Proceedings of the 16th National Computer Security Conference*. 1993, pp. 107–116.

[19] D. F. Ferraiolo and D. R. Kuhn. "Role-Based Access Controls". In: *Proceedings of the 15th National Computer Security Conference*. 1992, pp. 554–563.

[20] M. Fowler. *Strangler Fig Application*. June 2004. URL: `https://martinfowler.com/bliki/StranglerFigApplication.html`.

[21] M. Fowler and J. Lewis. *How to Break a Monolith into Microservices*. Apr. 2018. URL: `https://martinfowler.com/articles/break-monolith-into-microservices.html`.

[22] Z. Gao, C. Bird, and E. T. Barr. "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 758–769.

[23] E. Gilman and D. Barth. *Zero Trust Networks*. Manning Publications, July 2017. ISBN: 9781491962190.

[24] *Google Infrastructure Security Design Overview*. Google. URL: https://cloud.google.com/security/infrastructure/design/ (visited on 04/10/2020).

[25] *gRPC Documentation*. URL: https://www.grpc.io/docs/ (visited on 05/21/2020).

[26] N. Hardy. "The Confused Deputy: (Or Why Capabilities Might Have Been Invented)". In: *SIGOPS Oper. Syst. Rev.* 22.4 (Oct. 1988), pp. 36–38. ISSN: 0163-5980. DOI: 10.1145/54289.871709. URL: https://doi.org/10.1145/54289.871709.

[27] V. Hu, R. Kuhn, and D. Yaga. *Verification and Test Methods for Access Control Policies/Models*. NIST Special Publication 800-192. 2017.

[28] V. Hu, A. Schnitzer, and K. Sandlin. *Attribute Based Access Control Definition and Considerations*. NIST Special Publication 800-162. 2013.

[29] *Implementing Microservices on AWS*. White Paper. Aug. 2019. URL: https://d1.awsstatic.com/whitepapers/microservices-on-aws.pdf.

[30] *Istio Documentation v1.7.3*. Istio. URL: https://istio.io/latest/docs/.

[31] M. Jones. *JSON Web Algorithms (JWA)*. RFC 7518. Internet Engineering Task Force (IETF), May 2015. URL: https://tools.ietf.org/html/rfc7518.

[32] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515. Internet Engineering Task Force (IETF), May 2015. URL: https://tools.ietf.org/html/rfc7515.

[33] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. Internet Engineering Task Force (IETF), May 2015. URL: https://tools.ietf.org/html/rfc7519.

[34] M. Jones and J. Hildebrand. *JSON Web Encryption (JWE)*. RFC 7516. Internet Engineering Task Force (IETF), May 2015. URL: https://tools.ietf.org/html/rfc7516.

[35] A. Keromytis and J. Smith. "Requirements for Scalable Access Control and Security Management Architectures". In: *ACM Transactions on Internet Technology* 7 (June 2002). DOI: 10.1145/1239971.1239972.

[36] J. Kleeman. *We Built Network Isolation for 1,500 Services to Make Monzo More Secure*. Monzo. Nov. 2019. URL: https://monzo.com/blog/we-built-network-isolation-for-1-500-services/.

[37] J. Krach and W. Fu. *Open Policy Agent at Scale: How Pinterest Manages Policy Distribution*. Pinterest. Nov. 2019. URL: https://youtu.be/LhgxFICWsA8/.

[38] *Kubernetes Documentation v1.18*. URL: https://kubernetes.io/docs/home/.

[39] D. R. Kuhn, E. J. Coyne, and T. R. Weil. "Adding Attributes to Role-Based Access Control". In: *Computer* 43.6 (2010), pp. 79–81.

[40] *Kuma Documentation v1.0.0*. URL: https://kuma.io/docs/1.0.0/.

[41] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. Internet Engineering Task Force (IETF), July 2005. URL: https://tools.ietf.org/html/rfc4122.

[42] M. Leong. *Authorization at LinkedIn's Scale*. LinkedIn. Mar. 2019. URL: https://engineering.linkedin.com/blog/2019/03/authorization-at-linkedins-scale/.

[43] *Linkerd Documentation v2.x*. URL: https://linkerd.io/2/reference/.

[44] M. Loukides and S. Swoyer. *Microservices Adoption in 2020*. O'Reilly Media, Inc. July 2020. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/.

[45] N. Madden. *API Security in Action*. Manning Publications, Nov. 2020. ISBN: 9781617296024.

[46] G. Marquez and H. Astudillo. "Actual Use of Architectural Patterns in Microservices-Based Open Source Projects". In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 31–40.

[47] S. Meder, V. Antonov, and J. Chang. *Driving User Growth With Performance Improvements*. Pinterest. Mar. 2017. URL: https://medium.com/pinterest-engineering/driving-user-growth-with-performance-improvements-cfc50dafadd7/.

[48] A. Nehme et al. "Securing Microservices". In: *IT Professional* 21.1 (2019), pp. 42–49.

[49] S. Newman. *Building Microservices*. O'Reilly Media, 2015. ISBN: 9781491950357.

[50] S. Newman. *Monolith to Microservices*. O'Reilly Media, 2020. ISBN: 9781492075547. URL: https://www.oreilly.com/library/view/monolith-to-microservices/9781492047834/.

[51] J. Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 9780125184069.

[52] R. Nigmatullin and A. Ivanov. *Courier: Dropbox Migration to gRPC*. Courier. Jan. 2019. URL: https://dropbox.tech/infrastructure/courier-dropbox-migration-to-grpc/.

[53] P. Nkomo and M. Coetzee. "Software Development Activities for Secure Microservices". In: *Computational Science and Its Applications – ICCSA 2019*. Ed. by S. Misra et al. Cham: Springer International Publishing, 2019, pp. 573–585. ISBN: 9783030243081.

[54] *Open Policy Agent Documentation v0.23.2*. Styra. URL: https://www.openpolicyagent.org/docs/v0.23.2/.

[55] *OpenAPI Specification v3.0.3*. SmartBear. URL: https://swagger.io/specification/.

[56] M. Palladino. *Blowing Up the Monolith: Adopting a Microservices-Based Architecture*. Kong, 2020. URL: https://konghq.com/ebooks/adopting-microservices/.

[57] *Protocol Buffers Documentation*. URL: https://developers.google.com/protocol-buffers (visited on 05/21/2020).

[58] Q. M. Rajpoot, C. D. Jensen, and R. Krishnan. "Attributes Enhanced Role-Based Access Control Model". In: *Trust, Privacy and Security in Digital Business*. Ed. by S. Fischer-Hübner, C. Lambrinoudakis, and J. López. Cham: Springer International Publishing, 2015, pp. 3–17. ISBN: 978-3-319-22906-5.

[59] M. Richards. *Microservices AntiPatterns and Pitfalls*. O'Reilly Media, 2016. ISBN: 9781491963319.

[60] M. Richards and N. Ford. *Fundamentals of Software Architecture*. O'Reilly Media, 2020. ISBN: 9781492043454.

[61] C. Richardson. *Microservices Patterns*. Manning Publications, 2018. ISBN: 9781617294549.

[62] R. S. Sandhu et al. "Role-Based Access Control Models". In: *Computer* 29.2 (1996), pp. 38–47.

[63] B. Scholl, T. Swanson, and P. Jausovec. *Cloud Native*. O'Reilly Media, Aug. 2019. ISBN: 9781492053828.

[64] *Secure Development and Deployment Guidance*. The National Cyber Security Centre. URL: https://www.ncsc.gov.uk/collection/developers-collection (visited on 04/13/2020).

[65] A. Singleton. "The Economics of Microservices". In: *IEEE Cloud Computing* 3.5 (2016), pp. 16–20.

[66] P. Siriwardena. *Advanced API Security: OAuth 2.0 and Beyond*. Apress, 2019. ISBN: 9781484220504.

[67] P. Siriwardena and N. Dias. *Microservices Security in Action*. Manning Publications, July 2020. ISBN: 9781617295959. URL: https://www.manning.com/books/microservices-security-in-action.

[68] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel. "The Pains and Gains of Microservices: A Systematic Grey Literature Review". In: *Journal of Systems and Software* 146 (2018), pp. 215–232. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.09.082. URL: http://www.sciencedirect.com/science/article/pii/S0164121218302139.

[69] *Speedle Project Homepage*. Speedle+. URL: https://speedle.io/ (visited on 09/20/2020).

[70] C. Sridharan. *Distributed Systems Observability*. O'Reilly Media, 2018. ISBN: 9781492033424.

[71] C. Stivers and N. Higgins. *Deploying Open Policy Agent at Atlassian*. Atlassian. Nov. 2019. URL: https://youtu.be/nvRTO8xjmrg/.

[72] Y. Sun, S. Nanda, and T. Jaeger. "Security-as-a-Service for Microservices-Based Cloud Applications". In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 2015, pp. 50–57.

[73] D. Taibi and V. Lenarduzzi. "On the Definition of Microservice Bad Smells". In: *IEEE Software* 35.3 (2018), pp. 56–62.

[74] D. Taibi, V. Lenarduzzi, and C. Pahl. "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation". In: *IEEE Cloud Computing* 4.5 (2017), pp. 22–32.

[75] *The Modern Firewall: Reducing The Attack Surface, Securing Applications*. Forbes Insights with VMware Security. June 2019. URL: https://www.forbes.com/sites/insights-vmwaresecurity/2019/06/13/the-modern-firewall-reducing-the-attack-surface-securing-applications/.

[76] C. de la Torre, B. Wagner, and M. Rousos. *.NET Microservices: Architecture for Containerized .NET Applications*. Microsoft Corporation, 2020. URL: https://aka.ms/microservicesebook.

[77] E. M. Uchitelle. *Upgrading GitHub from Rails 3.2 to 5.2*. GitHub. Sept. 2018. URL: https://github.blog/2018-09-28-upgrading-github-from-rails-3-2-to-5-2.

[78] *Using Express Middleware*. URL: https://expressjs.com/en/guide/using-middleware.html (visited on 10/10/2020).

[79] M. Villamizar et al. "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures". In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2016, pp. 179–182.

[80] J. Vollbrecht et al. *AAA Authorization Framework*. RFC 2904. Internet Engineering Task Force (IETF), Aug. 2000. URL: https://tools.ietf.org/html/rfc2904.

[81] K. Westeinde. *Deconstructing the Monolith: Designing Software that Maximises Developer Productivity*. Shopify. Feb. 2019. URL: https://engineering.shopify.com/blogs/engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity/.

[82] E. Wolff. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016. ISBN: 9780134650449.

[83] T. Yarygina and A. H. Bagge. "Overcoming Security Challenges in Microservice Architectures". In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 2018, pp. 11–20.

[84] D. Yu et al. "A Survey on Security Issues in Services Communication of Microservices-Enabled Fog Applications". In: *Concurrency and Computation: Practice and Experience* 31.22 (2019). e4436 cpe.4436, e4436. DOI: 10.1002/cpe.4436. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4436.

[85] *Zero Trust: The Modern Approach To Cybersecurity*. Forbes Insights with VMware Security. June 2019. URL: https://www.forbes.com/sites/insights-vmwaresecurity/2019/06/12/zero-trust-the-modern-approach-to-cybersecurity/.

Appendix A

Authentication middleware

```
def requires_authentication(f):
    @wraps(f)
    def wrap(*args, **kwargs):
        user = get_user_or_none(request)
        if user is not None:
            if kwargs is not None:
                kwargs['user'] = user
            return f(*args, **kwargs)
        <...>
    return wrap

def get_user_or_none(req):
    try:
        token = req.cookies['accessToken']
        payload = jwt.decode(token, key=SECRET_JWT, algorithms=['RS256'])
        return User(payload['firstName'], <...>) if payload else None
    except KeyError: # The cookie is missing.
        return None
    except jwt.ExpiredSignatureError: # The token has expired.
        return None
    except Exception:
        return None

@exercises_bp.route('/<eid>/check/<pid>', methods=['POST'])
@requires_authentication
def evaluate_exercise(eid, pid, user):
    <...>
```