Tampere University

Joel Suomalainen

DEFENSE-IN-DEPTH METHODS IN MICROSERVICES ACCESS CONTROL

Faculty of Information Technology and Communication Sciences
Master's Thesis
02/19

ABSTRACT

JOEL SUOMALAINEN: Defense-in-Depth Methods in Microservices Access Control
Tampere University
Master of Science Thesis, 79 pages, 0 Appendix pages
February 2019
Master's Degree Programme in Information Technology
Major: Data Security
Examiners: Professor Billy Bob Brumley, Professor Davide Taibi

Keywords: microservices, security, authentication, authorization, service mesh, access token

More and more application deployments are moving towards leveraging the microservice paradigm in hopes of increased efficiency of operations and more flexible software development. Microservices are not a straightforward successor of existing methods and they introduce a lot of new complexity. Especially security concerns lack analysis in academic literature and new developments have mostly been assessed in grey literature.

The thesis explores the solutions to increase the security of microservice applications hosted in virtual private clouds. We start with the assumption that the networking security controls have been bypassed and the adversary is inside the network. We look at the situation through a holistic lens to identify the biggest gaps and how they can be filled in REST service-to-service communications. The solutions are platform agnostic to support the multi-cloud paradigm to reduce operational costs and increase global coverage.

Defense-in-depth methods proposed are establishing mutually authenticated TLS connections between services comprising an application and introducing granular access control using cryptographically secure methods. The industry state of the art ways to achieve these are assessed and analyzed comparatively and against good security engineering design principles. Both methodologies and their practical implementations are explored. We assess two distinct models for reference use for secure architecture design in microservices. These models piece lower level pieces into a comprehensive idea of what good microservice security looks like. The architectures can be used as is, as a basis for designing secure application architectures.

The thesis introduces security analysis of existing methods of deploying and establishing secure microservice applications, from container level orchestration to high level architectural choices. The work adds to the existing body of knowledge by assessing some of the security concerns enterprises moving towards microservice deployments are facing and by providing a new analysis of industry developments that have not been looked at thoroughly through a security lens in scientific literature.

TIIVISTELMÄ

JOEL SUOMALAINEN: Syväpuolustus mikropalveluiden pääsynhallinnassa
Tampereen yliopisto
Diplomityö, 79 sivua, 0 liitesivua
Helmikuu 2019
Tietotekniikan DI-tutkinto-ohjelma
Pääaine: Tietoturvallisuus
Tarkastaja: Professori Billy Bob Brumley, professori Davide Taibi

Avainsanat: mikropalvelut, turvallisuus, autentikointi, autorisointi, palveluverkko

Yhä useammat ohjelmistokäyttöönotot liikkuvat kohti mikropalveluarkkitehtuurin hyödyntämistä, toiveissa hyötyä tehokkaammasta käytönhallinnasta ja joustavammasta ohjelmistokehityksestä. Mikropalvelut eivät ole olemassa olevien paradigmojen suora jatke, vaan ne tuovat mukanaan uusia haasteita. Erityisesti tietoturvahaasteita ei ole analysoitu riittävissä määrin tieteellisessä kirjallisuudessa, vaan uusimpia kehitysaskeleita on lähinnä käsitelty yritysten tuottamissa julkaisuissa.

Tämä diplomityö kartoittaa ratkaisuja tietoturvan edistämiseksi pilvipalveluiden virtuaaliverkoissa ylläpidetyissä mikropalvelusovelluksissa. Aloitamme oletuksesta, että verkkoturvallisuuskontrollit on päihitetty ja hyökkääjä on verkon sisällä. Katsomme tilannetta kokonaisvaltaisen linssin läpi tunnistaaksemme isoimmat tietoturva-aukot ja kuinka ne voidaan tilkitä palveluiden välisessä REST-kommunikoinnissa. Ratkaisut ovat alustasta riippumattomia, jotta useiden pilvi-infrastruktuuritarjoajien samanaikainen hyödyntäminen on mahdollista operaatiokustannusten pienentämiseksi ja luotettavuuden parantamiseksi.

Esitellyt syväpuolustusmenetelmät ovat molemminpuolisesti autentikoidun TLS-yhteyden luominen saman sovelluksen mikropalveluiden välille, sekä yksityiskohtaisen pääsynhallinnan käyttäminen kryptografisesti turvallisten menetelmien avulla. Yritysmaailman viimeisimmät menetelmät näiden saavuttamiseksi arvioidaan vertaavalla analyysillä sekä niitä peilataan tunnettuihin hyviin tietoturvaperiaatteisiin. Arvioimme sekä metodologioita että niiden käytännön toteutusten tietoturvaominaisuuksia.

Tuloksena esittelemme kaksi mallia referenssikäyttöön tietoturvallisen mikropalveluohjelmiston arkkitehtuurisuunnittelun pohjaksi. Mallit yhdistävät matalamman tason osatekijät yhdeksi kokonaisvaltaiseksi ajatukseksi siitä miltä turvallinen mikropalveluarkkitehtuuri näyttää.

Diplomityö tuotti uutta analyysiä olemassa olevista metodeista tietoturvallisten mikropalvelusovellusten käyttöönotosta konttitasolta korkean tason arkkitehtuurisiin valintoihin. Työ lisäsi myös alan tietoutta arvioimalla yritysnäkökulmasta tärkeitä tietoturvahaasteita ja miten ne voidaan selättää. Tämä osaltaan paikkasi aukkoa akateemisessa kirjallisuudessa mikropalvelusovellusten tietoturvan osalta.

PREFACE

First, I wish to thank professor Brumley for expert guidance and pointers on how to turn my ramblings into a cohesive and presentable thesis. Also, thanks to professor Taibi for serving as the second examiner for my thesis.

I want to thank my colleagues in the application security engineering team at Riot Games for encouragement and initial sparring on whether anything I was saying made any sense. Especially the various ad-hoc discussions turned out to be an unexpectedly valuable resource for piecing bits and pieces together.

I wish to also thank my friends and family for helping me alleviate stress throughout the writing process. Special thanks to my girlfriend Johanne for offering relentless moral support.

Thank you, Haruki Murakami, for creating the perfect worlds to escape my own writing to. Some whisky, pasta, and an old recording of a jazz standard are surely in order.

In Dublin, 26.02.2019

Joel Suomalainen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ABAC	Attribute Based Access Control
ACL	Access Control List
AES	Advanced Encryption Standard
API	Application Programming Interface
CA	Certificate Authority
DevOps	Development Operations
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signing Algorithm
EdDSA	Edwards-Curve Digital Signing Algorithm
HMAC	Hash Message Authentication Code
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
MAC	Message Authentication Code
mTLS	Mutual Transport Layer Security
NGINX	Web server used as a load balancer and proxy
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OS	Operating System
OSI	Open Systems Interconnection
PKI	Public Key Infrastructure
RBAC	Role Based Access Control
REST	Representational State Transfer
RFC	Request for Comments
RPC	Remote Procedure Call
RSA	Rivest Shamir Adleman
SOA	Service Oriented Architecture
SSL	Secure Sockets Layer
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine

.

1. INTRODUCTION

The term microservice has not been formally established but microservices architecture is generally understood as a variant of service-oriented architecture that breaks a bigger, comprehensive application into smaller loosely decoupled components known as *microservices* (Bagge & Yarygina 2018). The goal is to provide better modularity, enable development of different parts of an application independent of others, and allow teams to employ the most suitable development, deployment, and testing strategies for their component (Richardson & Smith 2016 p. 9). The main benefits of microservices are outlined as isolation of issues, independent service scaling, and easier management of individual services (Microsoft 2017a; Dragoni et al. 2017). The loose coupling enables the different microservices to be developed with different technologies, as long as they have uniformly defined interfaces to communicate with each other (Dragoni et al. 2017).

When microservices are done correctly, they can provide a large amount of flexibility to development processes and are especially well aligned with the philosophy of continuous integration and deployment which have been adopted widely in the industry (Dragoni et al. 2017; Trihinas et al. 2018). The high degree of decoupling brings its own challenges from a security perspective and requires different threat modelling compared to monolithic services.

The gap between what industry leaders are doing now and the academic research was noted by Scoldani, Tamburri, and Van Den Heuvel (2018) in their paper "The pains and gains of microservices: A Systematic grey literature review" where they presented a systematic analysis on various industry driven publications on microservices through their lifecycle. The researchers noted in their conclusions that a lot of the pain associated with microservices in the design phase is due to the design of security policies (Scoldani et al. 2018). The lack of previous quality research has put researchers in a peculiar position, where more and more widely used industry practices have emerged, but much analysis on them is not openly shared and industry produced literature more resembles marketing material than credible research.

This thesis fits into the security niche of microservices and aims to answer some of the pains related to secure microservice application architecture and technology choices. The thesis contributes to analysis of some of the emergent security paradigms in microservices and provides a good basis for designing a practical microservice security architecture.

1.1 Research Questions and Scope

We set out to answer two research questions:

1. What are the defense-in-depth access control methods to protect a microservice application from an adversary inside the network?

2. What does microservice access control architecture look like with defense-in-depth security considerations?

We complement the research questions with some further constraints about the operating environment and adversarial capabilities we are defending against. The basis for the thesis is a situation where the traditional network perimeter defenses have failed and all of our service endpoints are exposed and we are relying on further defense-in-depth mechanisms to avoid further breaches.

The scope is focused on the analysis of existing security methods and piecing them into a cohesive distributed system. Cryptanalysis of the cryptographic methods powering the solutions is beyond the scope of this thesis. The thesis is also solely focused on service-to-service communications concerns and user-to-service methods are out of the scope of this thesis.

1.2 Solution Overview

To counter the adversary and establish defense-in-depth methods in microservice architecture communications, basic security objectives of confidentiality, authenticity, and integrity have to be met. Security measures can be implemented on several layers but an effective comprehensive solution requires thought put into security on each layer.

In this thesis, we establish through critical evaluation and comparative analysis, a holistic view into microservice architecture design that takes security into account on the container, service, and application level and counters the adversary with proven security methods. The analysis presented in this thesis may be used as design guidelines for secure architecture of microservice applications.

1.3 Structure of the Thesis

In section two, we look at the theoretical background of microservices and the security concerns associated with them to understand the holes in current knowledge. After we have established a suitable base knowledge of microservices, we will discuss some essential cryptographic concepts that power the security schemes discussed later. Additionally, we discuss the underlying cryptographic concepts to understand how access control methods provide the security guarantees they promise.

We discuss the used research methods in section three and define our adversarial model. Definition of adversarial model is required to understand the context of the thesis and understand the starting point.

In section four, we discuss how establishing strong cryptographic identities for services aids us in ensuring authentication in a zero-trust environment. We supplement learnings from academic research with knowledge gained from practical solutions to the problems that have emerged in the industry.

Section five explores how container orchestration and architectural patterns can be leveraged to provide stronger inter-service communications security. We analyze two widely used container orchestration systems and analyze how strong their security guarantees are for service-to-service context.

The sixth chapter focuses on different authorization schemes that build on the base we laid out with strong identities and authentication. We draft and analyze three authorization models that can offer further granular resources access control based on well-known security foundations.

In the last section, we combine the previously discussed factors in two comprehensive microservice system models, and assess their strengths and weaknesses through an architectural evaluation framework. Through this, we see what kind of operating environment and requirements each of them would best suit with the aid of a software architecture assessment framework based on the functional and qualitative requirements set for the system.

2. THEORETICAL BACKGROUND

The core idea of microservices has already been around for a while in the form of Unix principles of doing one thing and doing it well. From a slightly reductionist viewpoint microservice applications can be just viewed as an application consisting of isolated components working independently (Bagge & Yarygina 2018). One of the earlier applications of the term microservices aligned with the common understanding of them nowadays, was in the presentation "Micro services – Java, the Unix Way" by Lewis (2012), where he described a system consisting of small applications with narrow responsibilities communicating via a uniform web interface as microservices.

Alshuqayran, Ali, and Evans found in their paper "A Systematic Mapping Study in Microservice Architecture" (2016) that most of the existing microservice research was focused on research of evaluation or solution proposals. The researches also concluded based on the mapping study that the distribution of the types of research papers and evident lack of experience reports demonstrated that microservice research was still in its infancy in 2016 (Alshuqayran et al. 2016).

In academic literature, such as in the paper "Overcoming security challenges in microservice architectures" (Bagge & Yarygina 2018), microservices architecture is seen as an extension of the older Service Oriented Architecture (SOA) model. While this is arguably true on a high level of abstraction, the closer we look into microservices, the more unique characteristics we recognize that have not been adequately taken into account in the research on SOA security models before (Phan 2007; Davies et al. 2008 p. 225 – 264).

A container is a runtime instantiation of an immutable image that describes the requirements of the environment from the operating system (OS) to application dependencies (Souppaya et al. 2017). Immutability is important to ensure runtime consistency. Container environment consists of a host where the containers are run, using a shared kernel, as opposed to each of them running a separate OS instance as is the case with virtual machines (VM) (Souppaya et al. 2017). Containers allow for lighter use of computing resources on the host compared to VMs, spinning instances up and down gets quicker, and the penalty of hosting smaller services instead of monoliths lessens with lighter overhead. This makes them a great pair for microservices.

Movement from large scale deployment of virtual machines farms, to code defined containerization is tied to the increased popularity of development operations (DevOps) in the pursuit of infrastructure defined in code for greater automatization capabilities (Kang et al. 2018; Trihinas et al. 2018). As the development to live deployment loops get shorter, higher degree of decoupling of the components enables independent iteration capabilities for teams working on different components of the same application (Dragoni et al. 2017).

The great portability of containers also offers an easy way of horizontally scaling services by replicating service instances as containers across heterogeneous platforms (Dragoni et al. 2018), making it possible to leverage multiple public and private cloud environments to optimize for operational and financial efficiency. Different services benefit from different optimizations (Richardson & Smith 2016), which can be independently leveraged by deploying them on different hardware.

2.1 Confused Deputies and How to Unconfuse Them

The description of the confused deputy problem in a computing context dates back to Norm Hardy's story about a confused compiler in Operating Systems Reviews volume 22 (1988). The problem outlined by Hardy concerned a compiler that users could use by providing the compiler with a name of a file to receive debugging information about that file. The compiler also had a feature for collecting statistics about language feature usage. To enable the compiler to write to a file to collect the statistics, it was given a license to write files in its home directory. In the same directory, there was a billing information file. If a user supplied the name of the billing file to the compiler as debugging output destination, the compiler overwrote the valuable billing information with debugging information. The compiler acted as a deputy executing on behalf of the user with no knowledge of whether the user had the license for the operation or not. (Hardy N. 1988)

The confusion of the deputy rises from having conflicting sources of authority and committing acts with their own rights on behalf of another party without knowledge of their licenses. Trying to fix this problem is an instant source of increased system complexity. Instead of the compiler using its own license and the rights associated with that license to carry out the operation, the execution needs to be based on the requestor's license. To fix this, the compiler can be the one handling the authority and make decisions based on the requester and the target functionality. If the requester has the rights to do the requested action, the deputy will execute with its own license on the behalf of the requestor. Now the problem is how the requestor's identity is ensured and tamper-proofed. On the other hand, the authorization can be handled at the target. The intermediate deputy, in this case the compiler, carries with it the requester's identifier and the target file determines based on the original requestor whether the compiler can carry out this operation or not. This means that the target file carries the burden of maintaining access control to itself. Still the question remains, how can the target be sure that it is a legitimate request originating from the correct entity.

This example translates directly into the microservices world. The requestor, the deputy, and the target are entities that need to talk to each other to ultimately ensure a working application for the client. For example, the deputy can be an aggregating service that collects data from several "targets" that the requestors call in order to provide information to the end user. To alleviate some of the burden of the target service and the executor, we can establish a separate service that maintains the information around who should have

access to what. This is in the spirit of microservices, if the goal is to have separate services provide a service and do that one well. Instead of each one of the services handling and maintaining lists of access rights we want a separate authorization service to do the bulk of this work and have everything in one maintainable place. While the heavy lifting is done by this separate service we still need the services to be able verify identity and authorization of the requestor.

2.2 REST, Microservices and Security

There are several ways to architecture microservice communications and interfaces. The architectural design choices determine whether the implemented system will be able to fulfil the functional and quality attribute requirements set for that system (Costa et al. 2016).

One of the leading choices has been the Representational State Transfer (REST) paradigm (Costa et al. 2016). The concept was first introduced by Roy Fielding (2000) in his Doctoral Dissertation "Architectural Styles and the Design of Network-based Software Architectures". Other popular schemes are Remote Procedure Call architectures (RPC) (Richardson & Ruby 2007 p. 14), its Google developed variant gRPC (Louis 2015) and Simple Object Access Protocol (SOAP) (Microsoft 2003). The reason we choose to talk about REST is that it has garnered the widest use in the industry while SOAP has been heavily phased out in favor of REST and (g)RPC does not provide interoperability out of the box with the large portfolio of REST based APIs (ProgrammableWeb 2019).

Originally Fielding (2000) and later refined by Costa et al. (2016) define REST through six constraints that can be expressed as:

```
REST = (C – S,S,$,U,L,CoD)
```

The first expression $C – S$ represents **client-server** communication pattern, where separate clients request services from the server through network requests (Costa et al. 2016). Though the common model suggests that components are either clients or servers, in practical applications, especially in microservice architectures, many components serve both as a client and as a server in different phases.

Second constraint S describes **statelessness** of REST systems, which means that the server does not uphold any state between requests and all the necessary data must be included in the request-response sequence (Costa et al. 2016). For microservices this means easier horizontal scalability through replication (Costa et al. 2014) as the state does not need to be transferred and any instance that is spun up works identically to already running instances. The statelessness constraint rules out using session cookies for access control and requires the authorization credentials or a token carrying the required information to be passed along every request. Yarygina (2017) asserts that a stateless security

protocol is an impossibility without violating REST constraints. From the pure constraints perspective, even a resource such as a security token signing key is considered a resource with a state. We consider like Richardson & Ruby (2007 p. 90) that the adherence to application statelessness, when no per-user or per-session state is established, is enough to be aligned with this REST constraint and the resources on the server can and must have states.

Third constraint is *$* representing **cache** (Fielding 2000). This is to increase performance through decreasing the latency of server fetching the requested resources and delivering them to the requestor. From a security perspective caching requests including access token verifications for increased performance introduces the risk of the server acting on stale information, which potentially leads to a revoked or expired token being used to access protected server resources.

The fourth constraint *U* means a **uniform interface** across components and is the one defining constraint that separates REST from SOAP style services (Costa et al. 2016). Resources on the server are accessed through a URI, in pragmatic REST implementations this usually means an Application Programming Interface (API) endpoint that is called with the defined parameters using Hypertext Transfer Protocol (HTTP) verbs. For example, a call to fetch a list of users from a REST API endpoint could look like:

```
GET https://example.com/api/v1/users
```

When adhering to a good uniform design, the API endpoints follow an intuitive naming scheme and calling conventions.

The fifth constraint stands for **layered system** *L*, which means that an application architecture can be composed of several layers where no component sees beyond the layer they are interacting with (Fielding 2000). This design choice is to promote restricting system complexity and increase independence of services (Fielding 2000). Common microservice mentality seems to be at odds with this approach Fielding proposes. Instead of hierarchical layers, microservices go more towards flat network layers and promote extreme independence where none of the services are reliant on intermediate layers. This is very much in contrast to the data-flow like network with filter components and shared caches as discussed by Fielding (2000).

The last constraint of REST is **code-on-demand** *CoD* (Fielding 2000). This is an optional constraint where the logic implemented by user agents can be extended with code received from the service in the form of e.g. JavaScript (Costa et al. 2016). From a security perspective, this raises several concerns over possible exploits of compromised services serving malicious code along the requests. Accommodating this constraint requires client side implementation of execution capabilities for the received code. As in microservices world the roles of a client and server are often very fuzzy, a piece of malicious code being

distributed could have severe cascading effects, where in a blindly trusting environment the malicious code would be spread from service to service in a wormlike manner.

The existing multitude of implementations of REST adhere to these "pure" constraints to a various degree and an often-used collective term for these practical implementations is *RESTful services*. From our microservice perspective, the most important constraints for practical implementations are the communication pattern with requests and responses, statelessness, and uniform interfaces.

2.3 Security of Microservices

From the paper "A Systematic Mapping Study in Microservice Architecture" (Alshuqayran et al. 2016) we see that security concerns were not in the top microservice challenges considered in original research, found in only 3 of the 33 papers examined. This demonstrates a gap in security research in the field of microservices. The same paper also showed that the most common security research approaches were focused on solution proposals and opinions, with experience reports and evaluation research appearing less frequently (Alshuqayran et al. 2016). This can be seen as a sign of the immature stage of the field.

The paper "Overcoming security challenges in microservice architectures" by Yarygina and Bagge (2018) presents an overview of the security challenges within microservice architectures and discussing industry developments of Docker Swarm and Netflix public key infrastructure (PKI) solution. The analysis is supplemented by description of a microservice security framework by the researchers (Yarygina & Bagge 2018) to address the earlier identified security challenges. The paper provides a good springboard look into the state-of-the-art of microservice security and identifies some of the important findings from industry. Ultimately the paper lacks comparative analysis between the mentioned methods and does not describe other emergent architectural models such as the service mesh to address the challenges. This thesis extends the existing research from that perspective.

In general, the biggest challenge microservices introduce is more complexity. Instead of one monolith accessing a database, now you might have ten different services accessing a database and a lot of functions that before would have been handled internally are now exposed to some extent to the outside world. Contrasting microservices with a monolithic application can be a bit misleading as applications seeming monolithic outside are often comprised of highly modular parts inside the application (Bagge & Yarygina 2018). In essence moving to microservice thinking means a move from inter-process communications to inter-service communications over a network, which introduces more concerns from networking performance and security perspective (Microsoft 2017a). Securing microservices applications sets a lot of requirements for duplication of security into each service that before could be handled in one service.

Yarygina and Bagge (2018) present a decomposed view of microservice security layers. It closely models the classic seven-layer Open Systems Interconnection (OSI) model dividing the hierarchy into six different sections from hardware to orchestration. Presented below in **table 1** is a condensation of the threat surface and security concerns related to each layer, paraphrasing Yarygina and Bagge (2018). In this thesis, we will be focusing on the three upper levels of the stack, communication, services, and orchestration. By choosing this focus, we are placing inherent trust on the three layers below doing things securely.

Table 1. *Microservice Threats per Layer*

Layer	Threat surface
Hardware	Hardware bugs such as Meltdown and Spectre.
Virtualization	Isolation of services, sandbox escapes, hypervisor compromises.
Cloud	Cloud provider's control over resources, inherent trust issues, reliability of provider.
Communication	Eavesdropping, Man in the Middle, identity spoofing.
Application level	SQL injection, Cross Site Scripting, mis-implemented access controls, weak cryptography.
Orchestration	Malicious nodes in the service cluster, compromising service discovery or CA.

The security paradigm with microservices presents a foundational change from the traditional walled garden, in which the garden is our internal network that we trust, guarded by network security controls. The problem with this walled garden approach is, that when the wall is breached and a service compromised, there is nothing to stop lateral movement. With microservices it is not viable to rely on a security boundary solely defined by the network structure. The security boundary with microservices is the individual service and each of the microservices should be thought as exposed to the internet and security measures should be according to this. This does not make network segmentation and isolating applications to their own network an outdated and redundant concept but states that they are not enough. Network security controls should be the first line of defense but not the ultimate one. Our environments need to be treated as zero trust environments with an adversary already inside them. Setting the security boundary on the service level and

trusting no input implicitly are important factors in limiting lateral movement in the case a service instance is compromised (Yarygina & Bagge 2017).

Jander et al. (2018) tried to tackle the challenge of microservice security in their paper "Defense-in-depth and Role Authentication for Microservice Systems" where they assess mutual Transport Layer Security (TLS) as one of the approaches to achieve defense-in-depth beyond perimeter security. The researchers (Jander et al. 2018) assert that establishing mutual TLS is hard and requires custom application code but do not delve deeper into it. While this is true, we want to look into methods of establishing mTLS that can abstract away that implementation complexity from the application developers.

Fortunately, it is not only limitations and challenges when it comes to microservices and security. Even though the threat surface of an application expands and we have to rethink our perspective, the nature of microservices can help alleviate some security pain when done right. The core idea of isolation and small, contained, stand-alone services means that a microservice can focus on providing that functionality the best it can (Dragoni et al. 2017). This shrinks the codebase of a service and makes assessing the security implications of a service easier. Still, the actual amount of code in a microservice architecture might be bigger compared to monolithic application as some of the functionality is bound to be duplicated. But measuring lines of code is hardly ever an accurate representation of complexity or effort required for a review.

Due to the way microservices are usually deployed using containers, they can enforce immutability and requiring changes to cause a rebuild and redeployment (Souppaya et al. 2017). For example, Netflix utilizes a tooling set called the "Simian Army" that periodically and randomly simulates service and even partial core infrastructure failures, to develop extremely resilient automated deployment methods and mindset for deploying and operating services (Izrailevsky & Tseitlin 2011). This in turn means that a compromised service instance most likely will not suffer from persistent attacker presence and surviving redeployment requires a vulnerability in the layers below the container level.

Microservices can be built with various, heterogeneous, technologies as long as they expose a standard interface, which they should if they are adhering to REST constraints. Otterstad and Yarygina (2017) propose that due to system heterogeneity low-level exploitations can be prevented or at least made harder. In a sense, this is obviously true as the attacker would need to employ a larger set of exploits to attack services using different technologies. But it also has echoes of security by obscurity. The security of the system should not be reliant on the security measure being a secret. Or as Shannon's maxim states, "the enemy knows the system being used" (Shannon 1949). Also, there is always a chance that the heterogeneity can introduce unexpected problems into the system that can turn into security vulnerabilities. For example, it introduces complexity to patching of libraries and services and complicates the deployment pipelines. It also means that resources allocated into security are spread thinner and more cognitional complexity to

understand the system introduced. Security gained from system heterogeneity is a natural characteristic of a distributed system but not a meaningful security feature to aim for.

2.4 Key Cryptographic Concepts

As the thesis will discuss several security protocols that derive their security from well thought out use of cryptography systems and primitives, we will go through the key concepts required to understand the security implications of different schemes for authentication and authorization introduced later on in the thesis. Any new cryptanalysis of the presented primitives is wildly beyond the scope of this thesis. Instead, we rely on existing well-regarded research on the security implications of each cryptographic primitive and scheme.

Let's define few basic terms first. **Confidentiality** means keeping information known to only authorized entities. **Data integrity** means that information has not been tampered with. **Entity authentication** means verifying the identity of an entity whereas **message authentication** means verifying the origin of data. **Authorization** is defined as permitting an action and **access control** is simply limiting access to privileged entities. (Menezes et al. 1997 p. 3)

The importance of cryptography cannot be understated. In order to strengthen the security posture of our microservice system, we need solid ways of identifying, authenticating and authorizing entities. Achieving these goals on all of three of these layers require cryptographic methods with strong security properties. We will look at the basis for public-key cryptography and hash message authentication codes and what security properties they offer that can be applied in our system design considerations.

Different standardizing bodies give recommendations on how strong the schemes used need to be at minimum. These recommendations are subject to change when new cryptanalysis is released that finds weaknesses in the scheme or advances in solving the underlying hard mathematical problem are made. Advances in computing power can also mean that certain schemes should be phased out. We use recommendations given by the National Institute of Standards and Technology (NIST) in the United States and European Network of Excellence in Cryptology (ECRYPT). The security of a cryptographic scheme or protocol is measured in **bits of security**. Bits of security represent the amount of work required to break a scheme, measured in 2^N operations (NIST 2012). The security of that particular scheme is then N bits.

2.4.1 Public-Key Cryptography

Asymmetric cryptography, also known as **public-key cryptography**, means a cryptosystem where each entity has a public key e and a private key d corresponding to that. The security is based on d being infeasible to calculate from e. This property is based on

mathematical operations that are easy to calculate forward but very demanding to reverse. One of the main reasons for the success of public-key cryptography is that the public key does not need to be secret, participants only need to know that the corresponding private key is known solely by the intended party. The requirement for the public key is *authenticity* only. It is easier to provide the authenticity of public keys than the secure distribution of secure keys for symmetric encryption (Menezes et. al. 1997 p. 283). Asymmetric encryption is much more computationally intensive than symmetric encryption, which in turn means that typically asymmetric encryption is used to encrypt small messages such as credit card numbers or to establish symmetric keys through key negotiation schemes to be used for bulk encryption. (Menezes et. al. 1997 p. 283)

RSA, named after the inventors of the protocol Rivest, Shamir, and Adleman, is probably the most widely used public-key cryptography scheme, introduced in the paper "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (Rivest et al. 1978). The security of the scheme is based on finding the e^{th} roots for an arbitrary number modulo N. There is no efficient method known for this challenge, also known as the RSA problem. The most effective known method for generic solving of the RSA problem involves factoring the modulus N, which is considered infeasible for a sufficiently large integer N. Other solutions for generic solving of the RSA problem have been proposed but were demonstrated to be equivalent to the hardness of factoring (Aggarwal & Maurer 2008). (Rivest et al. 1978; Menezes et al. 1997 p. 283 - 286)

Another widely used asymmetric cryptography options is elliptic-curve cryptography (ECC), which is based on the hardness of the elliptic curve discrete logarithm problem. The set of points satisfying a particular mathematical equation is called an elliptic curve. ECC is based on these elliptic curves over a finite field. (Boneh & Shoup 2017 p. 595 – 597)

Both RSA and elliptic-curve cryptography are used for digital signatures and key agreement schemes and are present in the specification for Transport Layer Security (TLS) (Rescorla 2018). The minimum key sizes as per the ECRYPT recommendations in "Algorithms, Key Size and Protocols Report (2018)" for future proof cryptographic uses are 3072 bits for RSA and 256 bits for elliptic curve based methods, which are equivalent to around 128 bits of security (Smart 2018).

2.4.2 Hash Message Authentication Codes

A message authentication code (MAC) is a one-way hash function that is key-dependent with the purpose of providing integrity and authenticity of a message (Menezes et al. 1996). The authenticity guarantee also has an interesting characteristic, as noted by Krawczyk et al. (1997), a published breaking of a MAC scheme has no adversarial effect on previously authenticated information, unlike breaking an encryption scheme that would mean danger to all the data encrypted with that scheme. A one-way hash function can be

simply turned into a MAC by encrypting the produced digest with a symmetric key algorithm (Schneier 1996 Ch. 18.14).

Bellare et al. (1996) consider that the security of any MAC scheme is quantified by the success probability of an adversary breaking the scheme as a function of the valid MAC examples seen by them *q* and the available time *t*. The scheme is considered broken when the adversary can find a message *m* that they have not seen and the corresponding, correct authentication tag *a.* Additionally the MAC scheme should be resistant against a **chosen message attack** where the adversary gets to choose the messages instead of just observing valid **known messages** and the corresponding authentication tags. (Bellare et al. 1996)

Encrypting hashes with symmetric key encryption poses implementation challenges. In order to have a secure MAC that cannot be tampered with, we need a secure symmetric encryption algorithm and a secure implementation of that in addition to a good hashing function and a secret used as the key for the symmetric algorithm. This also introduces performance implications, as the encryption algorithm needs to be run in addition to running the hash function. To alleviate these pains, simpler scheme for creation of MACs using the one-way hash function with a key arose. Hash Message Authentication Code (HMAC) is a message authentication code that is based on the usage of keyed cryptographic hash functions (Krawczyk et al. 1997; Bellare & Krawczyk 1997). In the **table 2** below the design goals of HMAC are presented.

Table 2. *HMAC Design Goals*

Utilize readily available hash functions that perform well in software and their code is freely available
Simple key usage and key handling
Security of the mechanism is dependent on the underlying hash function that has a well understood cryptographic analysis
Allow the underlying hash function to be changed

Bellare et al. (1996) establish that the security of HMAC can be based on the cryptanalysis of Nested MAC (NMAC) presented by the researchers, as long as the underlying hash function's pseudorandom properties are strong enough. Which in turn means that the security of the HMAC scheme is based on the security of the underlying cryptographic hash function and the chosen key (Bellare et al. 1996).

With the increasing computing power available, cryptographic schemes that were once considered safe do not hold up anymore. Secure Hash Standard of Federal Information Processing Standards (FIPS) by NIST defines a family of Secure Hash Algorithms (SHA)

to be used in computer systems by governmental agencies in the US, which lists the seven approved algorithms *SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256* (NIST 2015). The NIST Secure Hashing Standard requires applications to use hashing functions with at least 112 bits of security (NIST 2015). The expected collision resistance strength is half the length of its hash value, with the exception of *SHA-1* which due to new cryptanalysis is indicated to be considerably weaker than the 80 bits of security the hash length would imply (NIST 2012). Thus, the weakest hashing algorithm that should be used in practice according to NIST is *SHA-224* with 112 bits of security. The ECRYPT Algorithms, Key Size and Protocols Report (2018) on the other hand suggests that the minimum should be 128 bits of security for applications with no concerns for compatibility with legacy systems, setting SHA-256 as the weakest allowed algorithm (Smart et al. 2018).

2.4.3 Digital Signing

In digital signing a digest of the message is produced using a cryptographic hash function and then combined with a private key of an asymmetric key-pair to produce a signature. The implementation of the *signing equation* depends on the protocol. Signatures over message digests are used to make the schemes more performant due to the general slow operations of asymmetric algorithms and to provide message integrity guarantees. The guarantees are similar to MAC schemes. Digital signatures also have the advantage of providing non-repudiation as the secret key used for signing is only held by one entity, instead of two or more entities as with MACs. The recipient verifies the signature by creating a digest of the received message sent along the signature and comparing that with the digest that is produced by a verification function taking the corresponding public key and the digest as input. It is required that both parties have agreed upon the used algorithms and the used implementations produce identical results. Commonly used signing algorithms are based on RSA or elliptic-curve cryptography. (Boneh & Shoup 2017 p. 672 – 673; Smart 2013 p. 223)

The advantage of elliptic-curve cryptography is that an order of magnitude shorter keys can be securely used compared to RSA. This allows ECC methods to be quite a lot more performant in their signing operations and key generation (Lauter & Stange 2008). Common digital signing algorithm using ECC is the Elliptic Curve Digital Signing Algorithm (ECDSA) (NIST 2013). Arguably an even more performant variation of ECDSA based on Twisted Edwards curves has been proposed by the name of Edwards-curve Digital Signature Algorithm (EdDSA) (Bernstein et al. 2011). The system is quite new and has not been considered in the ECRYPT reports nor the NIST reports yet and popular libraries lack implementations of it. Due to this we do not consider this signing option in this thesis, even though Boneh and Shoup (2017) note that the construct of the scheme does not suffer from an unexplained choice of parameter like some NIST developed ECC curves.

The 2048 bits of RSA roughly correspond to 112 bits of security that was outlined by NIST as sufficient level of security for use but represent a weaker scheme than 256 bit ECDSA, which is comparable to 3072 bit RSA (Table 2 Barker 2016; Smart 2018). The hash function used in the signature creation needs to offer at least the same number of bits of security as the asymmetric algorithm used to sign the digest to not weaken the scheme (Barker 2016). This means that for example the 256 bit ECDSA or 3072 bit RSA should be accompanied by SHA-256 or stronger function, offering 128 bits of security.

In **table 3** below we have presented the results of a simple performance benchmark done with Libre SSL 2.2.7 testing options on single core of Intel Core i5 2.3 GHz on MacBook Pro 13" 2017. The following results are the average of 10 runs for 10 seconds of 256 bit (ECDSA) signing and signature verification using the NIST Curve P-256 and RSA signing and verification operations with the key length of 2048 and 4096 bits. The benchmarking does not offer the 3072 bit RSA option for comparison. Still, we can see that the drastic difference in signing for the advantage of ECDSA, while the 2048 bit RSA enjoys an edge in verification operations. For 4096 bit RSA, we see that the verification speed has dipped below that of the 256 bit ECDSA scheme. For schemes of comparable strength, it seems that the edge of RSA for verification operations exists but is smaller.

Table 3. *Performance of ECDSA and RSA signing and verification*

	Sign	**Verify**	**Sign/s**	**Verify/s**
256 bit ecdsa (nistp256)	*0.0005s*	*0.0023s*	*1961.3*	*425.5*
rsa 2048 bits	*0.028266s*	*0.000806s*	*35.4*	*1240.9*
rsa 4096 bits	*0.191132s*	*0.002929s*	*5.2*	*341.4*

It is notable that if the use case for digital signatures in microservices is most typically signing tokens, the ratio of signings to verifications is heavily lopsided on the verification side. A singular token can be verified thousands of times in service-to-service communication if one is passed along every request. This means that for one signing operation tens of thousands of verification operations are done, which puts the performance emphasis on verification. Whether this performance difference is meaningful enough to impact the choice of algorithm depends on the size of the requests the tokens are passed along and through that how much of computational overhead the verification operation represents.

3. METHODS

The thesis explores the best practices and research around authentication and identity management of services in microservices architecture. The goal is to map out the state of the art defense-in-depth methods in service-to-service access control. Based on these we propose and evaluate methods of achieving a deeper, more comprehensive, level of security in microservice architecture.

We begin by defining the theoretical adversary we are trying to counter. The generic system we use as the baseline is based on the needs of a game service system. We then use literature both from the industry and academia to recognize the most important foundational methods and protocols to counter the defined adversary. The literature was collected largely by searching academic journal databases with the relevant keywords, such as *microservices, microservice security, cloud authentication,* and *distributed systems security*. The gathering of material was followed by reading abstractions and conclusions to gauge their relevance. Scientific literature was supplemented with relevant technical reports from organizations such as NIST and industry white papers from known organizations such as Google and NGINX.

The foundations are followed by starting to build secure service-to-service communications from the bottom up, starting from container orchestration progressing all the way to high-level system architecture. The solutions at each level of abstraction are evaluated based on their security promises and adherence to the greater system-level goals and contrasted with each other using comparative analysis.

The different models are evaluated using architectural analysis with criteria prioritized based on the identified needs of abstract multi-cloud operated high-performance microservice applications. Analysis of practical implementations for more quantitative research was foregone due to the results not necessarily generalizing well, as well as simply not having satisfactory big and complex microservice systems available for analysis without running into the problem of exposing too intimate system security details.

3.1 Assumptions About the System

The microservice application we are considering here is a latency-critical application, which comprises of several services providing a functionality, such as data aggregation or executing transactions. The application's quality of service is dependent on low latency to ensure user satisfaction. Due to this, application and the services in it need to be able to be deployed to a multitude of globally diverse public clouds and on dedicated bare metal resources in datacenters. The loads are also highly dynamic and thus the system must be automatically scalable for financial operating efficiency.

The services currently are hosted in virtual private clouds (VPC) where they are protected by security controls protecting the network boundary. As the services inside the network are trusted, no further security controls have been implemented and each entity inside the boundary can issue arbitrary requests to other entities and have the target service respond because of implicit trust. All traffic is unencrypted.

3.2 Adversarial Model

To flesh out the mitigation model for our model system, we need to know what we are trying to protect against. Thus, we model an adversary for the system. Adversary model maps out the possibilities that a dedicated adversary would have against a typical micro-service application deployment. The aim is to present a realistic adversary inside a breached network where additional defense-in-depth methods are required to prevent data exposure, lateral movement, and compromise of services.

The adversary is assumed knowledgeable of all the schemes deployed and their source code. The only thing the adversary does not know are the secrets and private keys of services. This adversary is based on the so called Dolev-Yao Threat Model, where the adversary is carrying every message and can impersonate any other entity to send messages (Mao 2002). The adversary is considered successful if they manage to pose as another legitimate service or craft a request to successfully access resources they should not be able to.

As we are modelling a situation where the adversary has bypassed the network security controls and is inside the network, we have to treat our network as a zero-trust environment. We assume trust in the cloud platform provider and all the hardware the services are deployed on. Computational boundedness is also an essential assumption to make as an unbounded adversary would break all of the cryptographic schemes in this thesis. The disruption of services with overwhelming traffic or other denial of service methods are considered out of scope. Here are the assumptions for the adversary:

1. The adversary is co-located in the same network as the microservice application

2. The adversary can eavesdrop on any service-to-service communication

3. The adversary may try and tamper with the service-to-service communication

3. The adversary can issue arbitrary requests to any service

4. The adversary is computationally bounded

5. The adversary has no knowledge of secrets or private keys of services but knows their public keys

4. SERVICE AUTHENTICATION

To counter the adversary defined in the methods section, we need to establish a secure encrypted communication channel between services to ensure confidential communications that cannot be eavesdropped on. To ensure this communication channel is established between the intended parties and an entity cannot be impersonated, we need to be able to authenticate other entities with great confidence. When the adversary is colocated in the same network, there is also a possibility to tamper with the messages even if they are encrypted, which means that we need methods to detect that a message has been tampered with. Lastly, we need message replay protection to counter an adversary from for example replaying legitimate user credentials to access to resources they should not be able to. In the **table 4** below we have presented the communications security objectives to counter the adversary and how to achieve them.

Table 4. *Security objectives of Microservice communications*

Communications security objectives	**Achieved by**
Authenticity	*Mutual authentication*
Confidentiality / Privacy	*Encrypted messages*
Integrity	*HMACs and digital signatures*
Replay protection	*Nonces, sequence numbers, and timestamps*

To highlight the need to assess deep level security concerns even in “secured” networks we can look into what the Edward Snowden leaks revealed in 2013. An illustration from the Top-Secret slides revealed that the National Security Agency (NSA) found a way to infiltrate into the internal networks of Yahoo and Google cloud (Gellman & Soltani 2013). The extraction of unencrypted data was made possible as the encrypted communications from the external internet were decrypted on a front-end proxy and passed along unencrypted inside the internal network. The reveal immediately prompted Google to encrypt their data-center links, the so called east-west traffic that before this had gone unencrypted.

The internet de-facto standard for communications with authentication, encryption, tamper proofed messaging is Transport Layer Security (TLS) over HTTP, known as HTTPS (Rescorla 2000). This can be used for interprocess communication as well in our system.

Other solutions such as messaging buses also exist but as we have chosen to focus on REST, we are solely concerned with communication over HTTP and securing that.

4.1 Establishing Strong Identities

To have authenticated users, we need to have something to authenticate them against. **Identity** is defined as a group of attributes that describe an entity (Linden 2017). For machines or services, we can have attributes such as the public-key associated with them or a domain name serving as identifiers. The identifiers task is to distinguish between similar services or more granularly, service instances. Especially for service instances, most of the attributes they have are shared which lends itself to confusion. Thus, there is a need for methods for instances to be assigned immutable attributes that provides them a unique identity which cannot be forged or impersonated by other instances.

Common authentication schemes are based on the basic factors of something you know (a secret such as a password), something you have (a physical code generator, mobile authenticator), and something you are (fingerprint, retina scan) (Linden 2017). When dealing with computers and services hosted on them, the biological factors and any factor requiring human input such as the use of physical authenticators cannot be utilized.

Passing credentials between services using HTTP basic auth header, while offering wide applicability and ease of adoption, (Fielding & Reschke 2014) has no implicit security mechanisms and the security is reliant on the underlying authentication scheme. Without further controls we do not know who is using the credentials as there is no identity directly tied to their use, and we have to rely on secondary methods such as internet protocol IP and Media Access Control (MAC) addresses to identify the entity, which are vulnerable to spoofing and cannot reliably be used for unique identification.

Thus, for establishing cryptographically strong identities and a basis for authentication, we use **certificates** as they are already a very established concept on the internet. The whole notion of trusted websites and secure HTTP are based on certificates through TLS over HTTP (Rescorla 2000, 2018). As defined by Boneh and Shoup (2017), “In its simplest form, a certificate is a blob of data that binds a public-key to an identity”. The association of a particular identity to a corresponding public-key is done by a **certificate authority** (CA) based on a **certificate signing request** (CSR). It is up to that CA then to verify the identity of the requestor by the means they deem appropriate. The CA creates a certificate based on the CSR and signs it with their private key. The security of the certificate is based on the strength and secrecy of the private key and the used signing operation algorithm. The certificate is verified using the public-key corresponding to the private key the CA used to sign the certificate. (Boneh & Shoup 2017 p. 552)

The most common standard for Public Key Infrastructure certificates is the X.509 standard defined in the Request for Comments (RFC) 5280 of Internet Engineering Task Force

(IETF) (Cooper et al. 2008). The certificates issued often have validity period of a year or more (Fu et al. 2018). But as noted by Topalovic et al. (2012) a certificate can become bad long before the expiration date, due to compromised signing keys, reissuance of that particular certificate, or other myriad of reasons. As it stands, the certificate revocation controls described in the standard (Cooper et al. 2008) have been ineffective and run into many problems in practice. These include problems updating devices, as for example Certificate Revocation Lists (CRL) rely on constantly updated lists of revoked certificates (Cooper et al. 2008). Alternatively, they introduce extra network delays by checking certificate status from a trusted party like with Online Certificate Status Protocol (Santesson et al. 2013).

The researchers Topalovic et al. (2012) introduced a certificate scheme based on short-lived certificates valid for only a few days that can be renewed based on a long-term certificate. In the case of certificate compromise, the impact is tied to the short validity of the certificate and further renewing can be stopped, instead of the certificate being valid for a year or more in the worst case. By choosing to rely on revocation by expiration, performance gains and reduced certificate complexity can be had with a possible security trade-off in the form of reduced revocation capabilities. Reliance on a trusted party to provide revocation information also introduces further coupling.

A similar method was presented by Bryan Payne of Netflix at USENIX Enigma 2016. Long term credentials are used to fetch short-term credentials are stored in the system level secure enclave, such as the Intel Software Guard Extensions (SGX) (Payne 2016). Intel SGX aims to offer computation with integrity and confidentiality even in an environment where every privileged system program is malicious (Costan & Devadas 2016). Though, relying on a vendor-specific solution goes against the abstraction thinking unless the same feature is available on all the used platforms. Additionally, a hamper was put on the trust on SGX with the release of the attack dubbed “Foreshadow” by Van Bulck et al. (2018) that presents an attack on the Level 1 cache of Intel CPUs that can potentially allow user processes to read OS kernel memory, extract information from the secure enclave, and enable malicious virtual machines to read memory belonging to other virtual machines on the same machine.

Vulnerabilities in x509 certificates have been demonstrated based on weak hashing algorithms used, such as CVE-2004-2761 based on the weakness of MD5 (NVD 2004). Besides this, attacks have been demonstrated on the validation side of certificates. Barenghi et al. (2018) found that commonly used TLS library OpenSSL with default settings had an exploitable parsing logic vulnerability that enabled the researchers to pass syntactically invalid certificates as valid. This highlights that implementation robustness is an important concern in addition to secure algorithmic choices and even widely used standard libraries can prove exploitable. Custom solutions even more so.

For manual certificate requests, various identity verification methods exist, from phone calls and emails to notarized documents (DigiCert 2018). For an auto scaling micro-service system where containers are constantly spun up and spun down, the certificates have to be provided automatically without manual human interference, rendering these methods unviable. Still the certificates cannot be provided at a whim or we risk undermining the reliability of our authentication methods. When moving into a short-lived certificate world, the certificate process needs to be automated to a high degree.

4.2 (Mutual) Transport Layer Security

We need to meet four communications security objectives to consider the communication channel we are establishing secure. They are confidentiality, authenticity, integrity, and replay protection. As we are considering REST over HTTP, the de facto standard for secure communications is HTTPS that promises to meet all of these four objectives.

The protocol consists of two layers, the handshake layer and the record layer. In the TLS handshake layer the client authenticates the server and the communicating entities establish session parameters and negotiate a shared secret using asymmetric cryptography. Once this identity has been verified, an encrypted and authenticated channel of communications is established between the client and the server and TLS record protocol is followed. (Rescorla 2018)

In the most recent TLS protocol version 1.3, the handshake was distilled into three steps. A simplified presentation of the handshake is as follows:

1. Client sends Hello to server along with the supported cipher suite and an extension containing a list of symmetric key identities and key exchange modes.

2. Server generates the symmetric session key based on the chosen key identity and exchange modes. Server sends Hello, the chosen cipher suite and the chosen key agreement algorithm, along with its certificate encrypted with the session key.

3. Client decrypts and verifies the server certificate and generates the session key based on the server response.

Communications from this point forward are encrypted with the symmetric session key. Message authenticity and integrity are guaranteed through the use of MACs.

(Rescorla 2018)

The assumptions of TLS characteristics and security properties provided by it are based on the assertions of the newest TLS Protocol version 1.3, RFC 8446 in the Standards Track of IETF (Rescorla 2018). TLS guarantees confidentiality by using asymmetric encryption in the handshake phase to counter eavesdroppers and a symmetric key encryption

in the record protocol (Rescorla 2018). Entity authentication in the classic TLS handshake is done only by client based on the server's certificate and its verification chain leading to a root CA that is either trusted or untrusted by the client. Integrity in TLS handshake is based on the use of message digests and authenticated encryption (Rescorla 2018). On the TLS record layer replay prevention is achieved by making the ciphertext output dependent on a sequence number within an authenticated encryption scheme, which is also used to provide message integrity, authenticity, and confidentiality (McGrew 2008). The options available with TLS 1.3 (Rescorla 2018) are the schemes AES-GCM (Bellare & Tackmann 2016) or ChaCha20-Poly1305 (Nir & Langley 2018), which both promise to provide all three.

As the protocol consists of a myriad of cryptographic schemes there is a lot of surface for security analysis. Both the algorithm choices and how they are combined into a protocol are important. The complexity of the protocol also lends itself to implementation difficulties that can lead to compromised security as we have seen with incidents like Heartbleed (NVD 2014). Analysis of the protocol has been done for example by Gajek et al. (2008) from the perspective of universally composable security. Cremers et al. (2017) on the other hand provide symbolic analysis of TLS protocol version 1.3.

In the mutual TLS (mTLS) variant, instead of just the client side performing verification, the server also authenticates the client. While this makes the communication channel mutually authenticated it introduces complexity by requiring clients to also obtain certificates. When the client and the server are both services, we get to service-to-service authentication with mTLS. While enabling server side verification is a minor concern from a protocol point of view, the operational and infrastructural burden of scaling the obtaining and using of certificates becomes bigger. The model of getting certificates from external CAs quickly becomes too rigid and infeasible. In the presentation "MTLS in a Microservices World" by Diogo Monica, the need for automatable and code-defined infrastructure in order to get to service-to-service mTLS was emphasized as vital (Monica 2016).

Below in **table 5** are the distilled benefits and disadvantages related to Mutual TLS based on an internally run Public key infrastructure (PKI). The pros are related to the security gains through TLS and using certificates for strong identity of services. The cons on the other hand are largely due to the increased operational load of running a PKI and effects of shifting responsibilities from big CAs to the organization itself. If the factors on the right side of the table can be assessed with automation and good orchestration tools, mTLS answers a lot of the security challenges present in the breached walled garden model.

Table 5. *Pros and Cons of Establishing Mutual TLS*

Benefits	Drawbacks
TLS is a widely supported standard with good library support	Increased system complexity, introduction of infrastructure requirements
Mutual authentication on top of other TLS security guarantees	Monitoring and revocation handling for compromised entities
Strong service identities based on certificates	Everything has a certificate, increased requirements for certificate management
Internal certificate authority removes costly external CAs from the equation	Infrastructure needs to support automated bootstrapping to ensure smooth operation
Internal PKI can be customized to fit organizational needs	You are now responsible for running a PKI

5. ESTABLISHING AUTHENTICATION

Environments with ideal conditions where theoretical models directly apply, unfortunately rarely exist. Businesses impose restrictions on engineering based on budget, time, personnel, and other resources and there is a lot of prioritization involved in how engineers spend their time. As security for most enterprises is not a revenue generator but a risk minimizer, security rarely can act as a main driver in big technological decisions which are also usually to some extent bound to the history of technological choices at that company. Few companies have the luxury of discarding older, legacy systems, and building completely new systems. The unfortunate reality is that often these legacy systems carry a lot of business value and porting them onto a new technology stack is far from trivial.

In the cloud-based heterogeneous microservices world, the selection of a cloud provider is a big factor in play. For example, looking at Netflix and Spotify who operate solely on a single platform provider (Amazon Web Services and Google Cloud respectively), this choice has enabled them to develop a mature ecosystem of tooling and subject matter expertise around that particular platform (Amazon 2018; Google 2016). A lot of this ability is due to the performance requirements of their products. In general, multimedia streaming services are more bandwidth reliant and more lenient on latency. This in turn means that the geographical datacenter positioning relative to the customers is not as critical as far as the product's competitive edge goes. On the other hand, real time competitive online gaming services are not as demanding as far as the bandwidth goes, but are very reliant on low latency to provide a good player experience. This sets a requirement for coverage of the player base with datacenters located in close geographical proximity.

Due to the strict requirements on availability, latency, and global coverage, game services benefit from being deployed on a variety of platforms, such as on public cloud services like AWS and Azure, on physical hardware in datacenters, and on dedicated game service infrastructure like Garena for the South-East Asia market (McCaffrey 2016). Multi-cloud deployments can also offer ways to arbitrage cloud computing costs and increase resilience by not relying on a single provider (Guerrero et al. 2018). One options for building tooling is to develop custom solutions for every platform, which in turn means the expenditure of a lot of engineering effort that cannot easily be replicated in other environments and making simultaneous multi-environment deployments harder.

The other option is to find solutions that can abstract the underlying infrastructure to more generic computing resources. In practice, this can mean writing abstracting wrapper platform on top of the infrastructure to enable simultaneous deployment to several environments. Building this abstraction requires a big initial engineering investment, especially if the existing platform specific tooling is not easily extensible and integration requires

hacky solutions. The emerging of container orchestration for multi-cloud deployments has alleviated this pain and allowed building solutions on top of these already available platforms, such as Kubernetes and Nomad (Guerrero et al. 2018). Justification for the effort is the promise of payback in the form of saving developers from deployment pain later down the road. For example, to handle this abstraction an option is to use a domain specific language to match generic requirements to lower level cloud specific management languages or API calls, as was done by Sousa et al. (2016). Relying on the platform specific security properties is complex and not easily made interoperable, which also makes platform agnostic solutions attractive.

5.1 Container Orchestration

We have seen that the industry is heavily moving towards microservices for increased operations and development efficiency (Kang et al. 2016). What we are missing is how security needs to be taken into account in arranging and deploying containerized microservices and how to achieve defense-in-depth security goals. The role of a container orchestrator is to schedule containers, manage their resources and support fluid deployment of containers. Good container orchestration allows us to define the desired state of a service and automatically bootstrap and auto scale this service (Guerrero et al. 2018). Container orchestrators also expose APIs that can be used to manage the containers in a secure way instead of having to resort to connecting directly to individual containers. Managing container clusters as an entity is used to maintain their immutable state and reduce attack surface of containers as laid out by Souppaya et al. (2017) in the NIST Application Container Security Guide.

First, we will have a brief look into how orchestration can be leveraged to provide a lot of automated secure-by-default features with minimal security consciousness required from developers leveraging it. This makes it an attractive solution to look at for solving the problem of encrypting service-to-service communications and providing strong authentication. We look into two of the more popular orchestrators, Kubernetes (Kubernetes 2018) and Docker SwarmKit (Docker 2018). Kubernetes is chosen as arguably the most used orchestrator which is also leveraged in more holistic platforms such as the service mesh Istio. Docker SwarmKit on the other hand is developed by the container technology developer Docker itself and it promises attractive security options that are automatically bootstrapped. Both projects aim to be more than container orchestrators but for all intents and purposes they are here considered as building blocks in the context of greater microservices application architecture. Both choices have a variety of options to be deployed on a wide range of public cloud providers or private cloud solutions (Osnat 2018).

A fair question to ask is why does the intra service communication matter, why not just focus on the service-to-service level. For one, the different nodes (container engine instances executing containers) may be running workloads with varying levels of sensitivity

and we want to ensure that eavesdropping or identity spoofing does not happen in container clusters. This is also important as to why establishing strong worker-to-manager trust is very important, as impersonation of a node could jeopardize sensitive workloads. Limiting nodes to observe only their own traffic and removing the possibility of stealing workloads is important to limit the effects of a compromised node.

To answer these concerns, the most important orchestrator promises are secure trust bootstrapping and introducing nodes to a cluster, issuing strong identities to nodes, and establishing automated mTLS connections between nodes. Especially mutually authenticated traffic between cluster members, as well as between services with secure-by-default posture have been highlighted as important criteria for choosing a container orchestrator (Souppaya et al. 2017). These are essential as defense-in-depth methods and ensuring confidentiality, authenticity, and integrity of service-to-service communications starting from the container host level.

5.1.1 Docker SwarmKit

Docker SwarmKit promises least privilege container orchestration with automatic by-default security guarantees (Docker Swarm 2018). In a cluster of containers hosting a service, the managers are responsible for scheduling and serve the Swarm API to maintain the defined state of the service. The worker nodes execute containers and do not schedule or maintain internal state of the swarm. Communication is done solely manager-to-node to narrow the threat surface. (Monica 2017)

When a swarm is initialized, the first node created is designated as a manager node and a new root Certificate Authority (CA) is generated or the node pointed to an existing CA. This CA is used to secure communications in the swarm between the nodes joining it. New nodes joining the swarm use a token designated to their role which is constructed of the digest of the root CA certificate and a randomly generated secret. The joining node validates the root CA certificate from the remote manager using the digest in the token. The remote manager validates the node is approved to join the swarm with the randomly generated secret included in the token, which is submitted to the manager as a Certificate Signing Request (CSR). The issued x509 certificate includes a randomly generated ID under the common name and organizational unit of the CA, which establishes an immutable identity for each node. The certificate type depends on the joining nodes role. Based on the x509 certificates nodes have from the same CA, they can establish mTLS connections using TLS 1.2 with other nodes. (Docker SwarmKit 2018; Monica 2017)

The importance of protected and immutable identities is to prevent compromised nodes from getting access to other workloads or unauthorized nodes from joining the cluster. The default interval for certificate renewal is three months but can be as short as thirty minutes (Docker SwarmKit 2018). The short renewal is to limit the effect of compromised

nodes and to serve as the revocation mechanism instead of utilizing for example revocation lists. Renewal of certificates is established via mTLS connection to the root CA and subsequently generating a new public/private key pair and a corresponding CSR which is sent to the root CA that provides the node with a new certificate encapsulating its identifiers (Monica 2017). A known compromised node would be blocked from getting a new certificate. This of course requires that the compromise was monitored in the first place.

Credential rotation is achieved by Docker generating a new root CA certificate that is signed by the old certificate (Monica 2017). By doing this the nodes that still trust the old root CA can validate certificates signed by the newly created root CA. In essence, the new root CA works as an intermediate CA in relation to the old one. All the nodes in the swarm are forced to renew their certificates immediately. After all the nodes have renewed their certificates the old root CA is forgotten by Docker and the intermediate CA becomes the actual root CA.

For node ID creation, the program uses cryptorand from the Go crypto library (Docker SwarmKit 2018). On Linux systems, this uses *getrandom* to get entropy from the urandom source or /dev/urandom that uses the same source for entropy (Golang Docs 2018). The difference between these two being that *getrandom* waits for enough entropy before generating numbers from system usage (Linux MAN 2017). Result is a fixed 25-character long identifier in base 36, which translates to a padded or truncated 128-bit number (Docker SwarmKit 2018). Though the robustness of /dev/urandom has been brought to question in its analysis (Dodis et al. 2013), it is still recommended as best practice in industry in practical applications when hardware generators are not available (Hühn 2018). The join token's only non-secret part is generated similarly to the node ID and acts as the safeguard to bar unauthorized nodes from joining the swarm, especially as managers.

The worst-case scenario is the compromisation of the swarm manager serving as the CA. This would allow it to grant access to the swarm to arbitrary nodes and breaking the boundary of trust. Deeper protection level of individual containers can be achieved by different hardening methods but they are out of the scope of this thesis. With Docker SwarmKit, secure container-to-container communications can be established. This allows confidential communications starting from the lowest level of our system based on mutually authenticated TLS and its security guarantees. The parts of the security analyzed provide the promised level of security guarantees with best practices, increasing confidence that the system can be trusted.

5.1.2 Kubernetes

Kubernetes, like Docker Swarm, is a container orchestrator for Docker containers. Originally developed by Google for deployment automation, scaling, and container management, Kubernetes is now an open-source project (Kubernetes 2018a). A basic unit of

management in Kubernetes is a pod which is a container or a set of containers hosting a service or an application comprised of services (Kubernetes 2018b). Services are comprised of a logical set of pods and an access policy to them (Kubernetes 2018c). Similar to SwarmKit it works towards maintaining the desired state defined in configuration files. In the version 1.12 Kubernetes introduced a feature called "Kubelet TLS bootstrap", where a node joining a cluster will generate a CSR for a cluster-level certificate authority, which will return a certificate for the service account associated with the particular service that is going to be run on that node (Tankersley 2018). Similar to SwarmKit, the revocation mechanism is the short life of a certificate. Kubernetes is also promising a feature in the future for server certificate bootstrapping and rotation, which require manual injection at the time of writing. With these additions, the container orchestration security comes to a similar level as with SwarmKit. But as it stands now, SwarmKit offers the more complete solution on this issue. Kubernetes development is done in Golang similar to SwarmKit and uses the same crypto/tls and crypto/X509 libraries to establish TLS connections and the certificates required (Kubernetes Community 2018c).

When it comes to identity management and authentication with Kubernetes, it is a mixed bag of insecure and secure options. The options include static basic auth, bearer tokens, and certificates (Kubernetes 2018a). One of the two recommended options by Kubernetes is using a 128-bit random token created with dev/urandom/ that is stored on the API server in a token list. The problem with this is that the tokens last forever by default and adding or removing tokens from the checklist requires a restart of the server (Kubernetes 2018d). The other option is using a "bootstrap token" which is a signed JSON Web Token using SHA256 HMAC and a shared secret (Kubernetes Community 2018a). The token ID associated with the token is only 6 characters and thus highly guessable. As the secret is fetched based on the name of the secret and node from the "manager", this is a vector for system compromise. Essentially a joining node fetches a token from the API that it uses to authenticate to that API to create a CSR. This risk was recognized by the designers themselves and to thwart it the default lifetime of a token is 24 hours to limit the effect of compromise (Kubernetes Community 2018a).

5.2 Comparison of Container Orchestration

Kubernetes is an extensive platform and offers a lot of options when it comes to security and also a lot of room for developers to make insecure decisions as all the options are not default-by-secure. SwarmKit, while the more limited platform general capability wise, embraces the default-by-secure ideology in a more fool-proof way. Its automatic security bootstrapping abstracted away from the developer leads to less decision making burden and reduced risk of making insecure choices.

Both options provide the security properties we wanted from them, but in different ways. These different strengths have been recognized in the industry and for example the Docker Enterprise Edition supports both Kubernetes and SwarmKit (Hykes 2017).

SwarmKit can be used to establish lower level security to a Kubernetes deployment, which on the other hand has more extensive management capabilities through its API (Fisher 2018).

The **figure 1** below illustrates a generic service constisting of worker nodes and a manager hosting the cluster control API. All the connections inside a container cluster are protected by mTLS. This is the default deployment state of a SwarmKit deployment and it can be also configured to happen with Kubernetes.

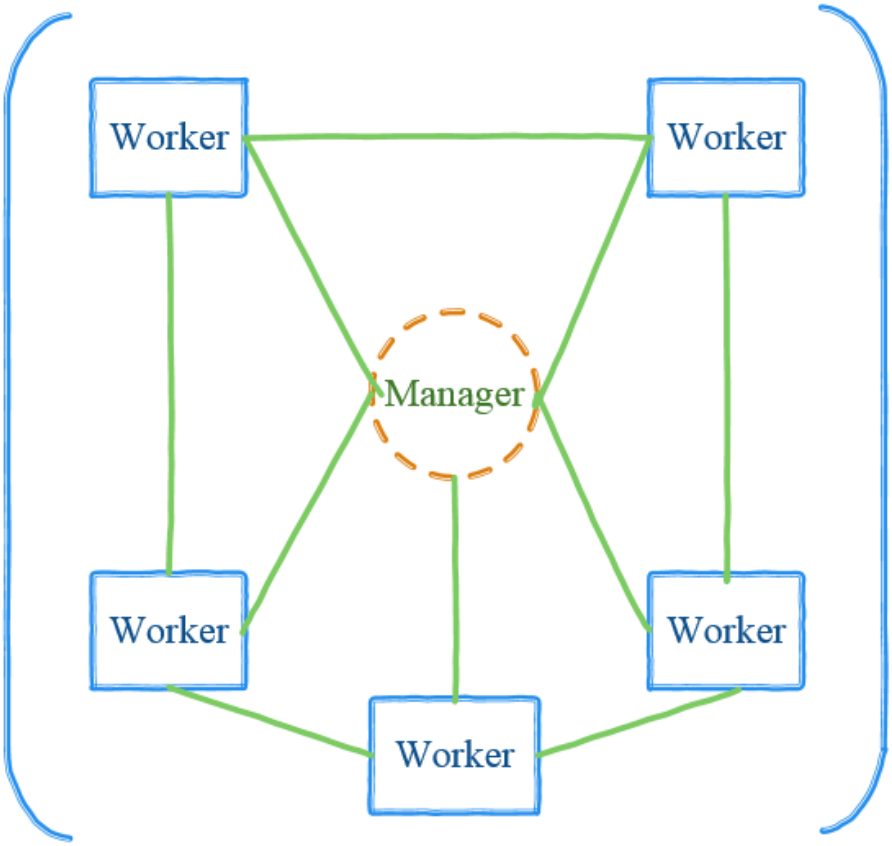

Figure 1 Service Comprising of a Node Cluster

5.3 Microservice Patterns

We have now established two ways container orchestration can be used to provide secure container-to-container communications inside services. These clusters of containers form our basic building block, a service. We will now look at how these services can be arranged according to different microservice patterns. We now make the assumption that the underlying container-to-container communication is confidential, authenticated, and message integrity is guaranteed. The different pattern approaches will be evaluated based on their security properties and complexity of their implementation.

The sophistication of the various patterns varies from a simple security gateway to a multi-layered service mesh solution. We focus on the security they can provide and with how much complexity and implementation baggage they come with in order to achieve the sought properties. We start by looking at what kind of patterns exist at the container level and move up in abstraction and complexity, assessing proxy gateway, proxy mesh, and lastly the service mesh model. The choice to assess these three models is based on their adaptation in the industry and the Microservices Reference Architecture report by Stetson (2018).

5.3.1 Single Node Patterns

Sidecar is a container colocated with an application container that provides or extends some of the functionalities of the application (Microsoft 2017b). Example use of a sidecar is to function as a proxy that provides TLS capabilities to a legacy application by terminating the TLS traffic on it and providing unencrypted traffic to the application container through localhost on the same machine (Burns 2018). Other functionalities can include things such as service discovery, logging, and load balancing depending on the greater architecture and the division of labor. If used in this manner as a sidecar proxy, the application container is only aware of its local proxy and not the greater network beyond it (Klein 2017). Even application level security such as authorization controls can be implemented here, such as enforcing authorization policies.

The decomposition of features into a sidecar allows them to be developed separately from the application (Burns 2018). This allows it to be developed by subject matter experts in that particular area, such as networking, leaving the application developers to work on the core differentiating functionality. Sidecars can also be developed with different technology than the main application and can be consumed by applications written in various ways, as long as the interfaces are standardized (Microsoft 2017b).

Similar to a sidecar, the Ambassador pattern means tight coupling with a single application container. Its purpose is to broker connections to the outside world from the application container through localhost (Burns 2018 p. 14). An example of this is providing service discovery to the application container. The third pattern described by Burns (2018) is the adapter model, which is used to modify the interface of the colocated application container to conform to a generic system standard e.g. for logging. In grey literature, all the single-node patterns are often mixed and for example when Envoy sidecar proxy is discussed, the features it provides can be associated with Ambassador, Adapter, and Sidecar patterns.

The potential lies in being able to provide security properties to applications without having to having to implement them per-service in application code. Implementation difficulties of mTLS can be passed onto the supporting container, which can be developed independently of the service and generalized to be used by a large amount of services.

The paradigm is aligned with general microservice idea of services that focus on their main functionality, as now the common non-differentiating security and networking features can be provided in a container that can be automatically deployed alongside the main service. Some of the complexities are offloaded to operations and container orchestration which need to ensure successful deployment.

5.3.2 The Proxy/Gateway Model

In the proxy gateway pattern, requests to various microservices are delegated through a gateway proxy that further delivers them to the correct service (Stetson 2018). This pattern is mostly used in client to service interactions with an API gateway rather than service-to-service calls. The reasons for this are tied to the performance hindering effect. In a service-to-service communications context this means all the requests would need to pass through this gateway to direct the requests to the correct service. This automatically makes the proxy gateway a single point of failure and often the performance bottleneck. An example of the model is presented in the **figure 2** below.

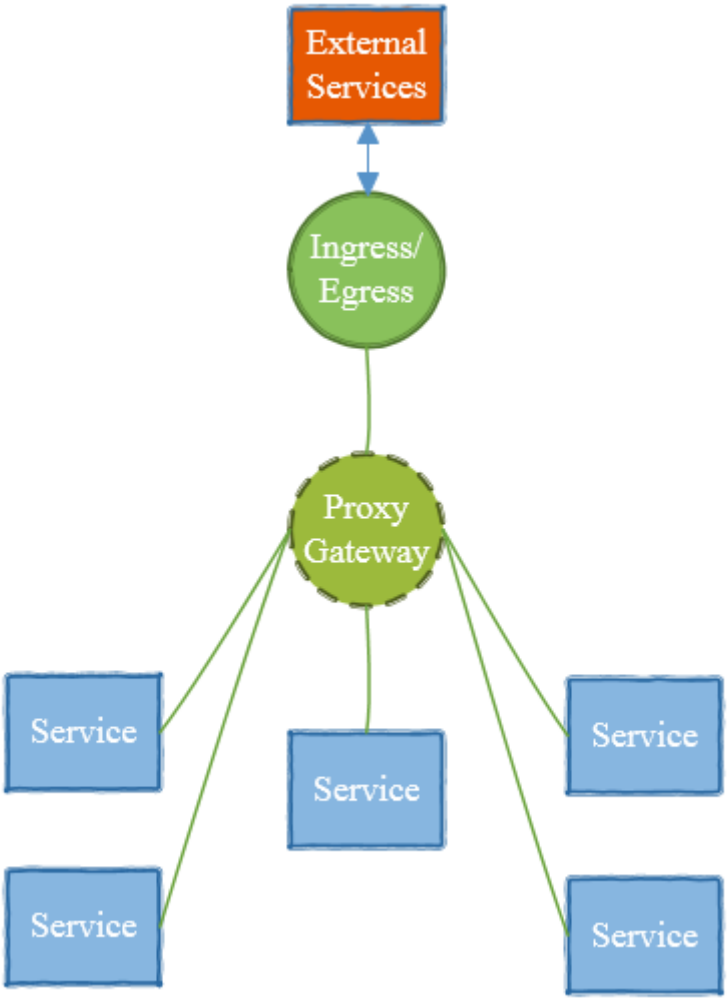

Figure 2 The Proxy Gateway Model Illustrated

A proxy gateway based solution was proposed as a solution for scalable data management to handle large amounts of environmental time series data in the research by Braun et al. (2017) presented in the paper “A Generic Microservice Architecture for Environmental Data Management”. The description of the model did not take any particular considerations for security and asserted that the inter-service and service-to-gateway communications is done over HTTP(s) but did not provide any solutions as to how that would be achieved (Braun et al. 2017). The analysis in favor of this particular architecture was mainly reliant on the advantages of microservices and application decomposition in general and was no comparative analysis was provided to alternative microservices architectures.

Yarygina & Bagge (2018) also presented a framework using an API gateway combined with tokens for user principal delegation inside the system. It leveraged mTLS for securing service-to-service communications with an internal PKI but its focus with tokens was more on securing external user interactions with the services and principal propagation. The system as a basic idea covers a lot of the security basics but the description lacks detail as to how the properties are achieved in practice, which the authors Yarygina & Bagge (2018) recognized as well.

The approach with a security enforcing gateway does little to discourage the developer from resorting to the old walled garden approach. When we put a gatekeeper in front of our services to handle security, we implicitly say that we trust everything beyond the gatekeeper. This goes against the zero-trust principle of operation that we are striving towards with microservices security.

Proxy and API gateways can coexist with other better suited methods for inter-service communications. In these types of systems, the API gateway can serve exclusively to handle external third-party connections to the microservices that intend to expose and external APIs, while not interfering with the communication patterns of the services. The model can also work for applications with no or very little inter-service communication required. In any case, the communications between services and the gateway should still be secured using mTLS. The sidecar pattern can be used to achieve this but the more networking functionality is offloaded to the sidecar the more it starts to make sense to forego the proxy gateway model.

5.3.3 Proxy Mesh

In the proxy mesh model, as presented in the Microservices Reference Architecture report (Stetson 2018), we establish a central inter-service proxy or proxies to handle internal service-to-service communication. The cluster of proxies can be scaled to service larger

microservice infrastructures. For external communications, there is a second ingress/egress proxy cluster that relays the connections to the inter-service proxy that further distributes the request to appropriate services (Stetson 2018).

The advantage of this approach is that services can rely on the inter-service proxy to provide service discovery, load balancing, and service health checks (Stetson 2018). When TLS termination and establishment is done on the proxy we get guarantees of encrypted and authenticated communications to external services. To guarantee encrypted internal communications, mTLS connections still need to be implemented between the service and the proxy. Thus, every service needs to implement mTLS capabilities that are uniform with the proxy. If services are developed with different technologies a single library cannot be just used necessarily and developers need to duplicate efforts. The proxy mesh can also be combined with the sidecar pattern to provide services TLS capabilities. At that point benefits of the proxy mesh are limited to load balancing and possibly service discovery if the sidecar does not also provide these capabilities. This adds an extra hop as network overhead when all the service-to-service calls go through the proxy to the target service, even in the same network.

In the **figure 3** below we see a simple representation of a proxy mesh architecture with one inter-service proxy handling internal service-to-service requests for the six services comprising an application and passing requests to proper services originating from external services. For proper defense-in-depth security all the connections in the model should be protected with the use of mTLS.

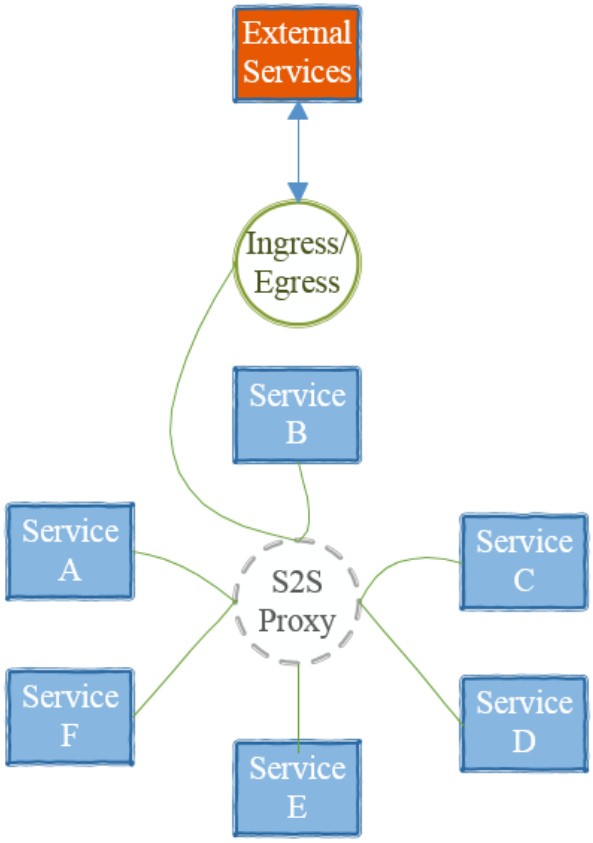

Figure 3 Illustration of the Proxy Mesh Model

Scaling the solution is achieved by replicating instances of the proxy itself and forming larger clusters. The performance of the services themselves are also horizontally scalable by replication and creating larger clusters to host the services. The problems of the model are related to how the individual services achieve mTLS and authorization capabilities. The model does not offer a generic and shareable solution to this challenge and the services still need to implement them in application logic, unless sidecars are used. If sidecars are used, it becomes a balancing question of which features we want to offload onto the colocated container and how much to a central proxy.

5.3.4 Service Mesh Model

A generic service mesh (also known as fabric) model consists of two layers or planes:

1. **Data plane**
2. **Control plane**

The sidecar proxy is akin to the single node patterns introduced in a previous section, where a colocated container extends the application container by providing features such as TLS capabilities, service discovery, and implementing access control logic. Sidecar

proxy has turned out to be a bit of a catch-all term, conflating existing distinct patterns into one. We conform to use the sidecar proxy term here, in order to be aligned with established industry terminology in microservice architectures.

In a service mesh system sidecar proxy works as the data plane which manages networking features such as service discovery, service health checks, and load balancing traffic (Klein 2017) and can be thought as a replacement for physical network routers and switches. As the logic for managing connections is done on this layer the security concerns are around managing mTLS connections and authorization logic. Available sidecar proxies include NGINX (NGINX 2018), Envoy (Envoy 2018), and Linkerd proxy (Linkerd 2018).

The highest level of abstraction and the plane that humans usually deal with in service mesh systems is the control plane. This is the layer where humans control the system. In a service mesh the purpose of control plane is to tie the different networking pieces and functions together into a distributed system that can be managed through software and automation. Examples of control plane solutions are Istio, Linkerd, and Consul. (Klein 2017)

Comprehensive service mesh solutions offer many other features besides mTLS, ranging from telemetry to automated configuration deployments and secrets management. From a service-to-service security perspective Istio offers “out of the box” mTLS connections and well-defined authentication and authorization policy options through the use of sidecar proxy Envoy. For Linkerd, automatic mTLS bootstrapping is still in experimental mode at the time of writing but is looking to be supported through their own proxy (Linkerd 2018). With Consul, native and sidecar proxy options are offered, with a Go library and the possibility for proxy injection (Consul 2018).

5.4 Pattern Comparison

We saw in the previous section that the actual implementation of the pattern does not have a direct effect on system security. But it mandates how generalizable the security solutions can be, the division of responsibility for security controls, and how much custom code is needed for the implementation of security controls.

In general, all of the aforementioned service patterns can offer the same security properties. When all the patterns are equal security wise, the one offering the most convenience and efficiency is the most alluring. For example, industry professionals have highlighted the difficulties associated with interoperability problems when using SSL, such as managing certificate trust stores, key delivery, and choosing the right algorithms to handle the actual protocol (Posta 2018). Abstracting these away from the developers is thus something to aim for.

Taking into account the usability issues, the service mesh pattern seems like the best bet. It enables the developer to dedicate the security concerns of establishing secured communications with authentication and authorization methods to a local sidecar proxy, thus abstracting all the concerns to another layer from the application logic. As long as a uniform interface is kept for communication between the service and the proxy, the code in both can be modified independently without having concerns about breaking interoperability.

Another option for performance critical applications where sidecar proxies present extra overhead is building around the proxy mesh idea with native security implementations on the services themselves. This comes with the overhead of replicating code and possibly more code development if the services are implemented with different technologies. In team with expertise in security matters the lack of generic implementations might not pose an issue, especially when performance requirements are very high. While the downsides are related to engineering effort the upside is in the possibility for greater optimization and application specific customization that can manifest in better performance and control over services.

6. SERVICE AUTHORIZATION

In the earlier sections, we found a way to establish a secure communications channel that offers confidentiality, authenticity, and message integrity with mutual TLS connections. This stops the adversary from eavesdropping, tampering with the messages, and being able to impersonate some other entity. Yet, authentication by itself is not enough for comprehensive protection as a trusted and authenticated service could still be compromised and send malicious requests to access resources beyond the legitimate needs of the service. Now we need methods to use authentication to mandate access to resources based on a policy. The goal after all is to limit the effects of a compromise to the compromised service by restricting all available access to only what is really required. For this we need methods for authorization.

The most naïve solution is to trust an authenticated service and fulfill their requests, whatever they are. If they need user A's info, the service would not limit them to access just that, but all the user's and trust the service to do the right thing. Problem is the services have no concept of right or wrong beyond their programmed logic. This leaves a door open for a malicious entity to craft requests in a way that could let them access resources not available to them, making the service a confused deputy. Either the requestor or target service needs to implement logic to gauge whether access is permitted or not. Though if the logic was implemented only on the requestor and it got compromised, the security control would be null. Thus, the logic needs to be implemented by the target service. Even then, poorly implemented logic would also leave room for a crafty malicious entity.

That is why we need further granularity in our access control schemes to enforce that clients are only accessing resources their legitimate use case requires. If we consider a traditional firewall, its purpose is to block all connections except the legitimate ones we approve of. Where this firewall approach fails is when a malicious request is coming from a legitimate source that has been compromised or tricked to make the request. For defense-in-depth we need an additional layer of security. Still the implementation of authorization does not remove the problem of a compromised service. If a legitimate service is compromised, it can always request access to the target service until the compromise is monitored and acted upon. But it does limit the effect of compromise.

With authorization controls we are leaning heavily on the principle of least privilege, which was described by Saltzer and Schroeder (1975) in their publication "The Protection of information in computer systems" as every program and user operating with the least privileges to perform their duty to limit the damage of a potential accident or error and reduce the risk of misusing privilege.

Schneider goes on to build on the early definition of the Principle of Least Privilege to assert that policies that adhere to the principle are not only considering the actual code to be executed but what job is the code intended to do (Schneider 1975). From the definition presented by Schneider (1975), we can derive the expression

$Priv_{min}(Exec, Done_{EXEC})$

where $Priv_{min}$ is the set of least privileges an actor needs to carry out *Exec* the job defined as done by $Done_{EXEC.}$

The challenge is in how to arrive at this set of least privileges and make sure the defined policy is enforced. Schneider (1975) did not come to a definitive answer as to how this could be achieved but suggested that a more feasible solution would be to look at policies that sought to prevent from violating the base level assumptions about a system, if the assumptions can be expressed as security properties.

The authorization schemes presented next do not answer the question of how to arrive at the set of least privileges required, but rather offer tools to guide policy enforcement and design of practical schemes. What the least privileges set looks like is heavily dependent on the application and thus a generic definition beyond that of Schneider's (1975) is not possible to arrive to. We will discuss and analyze a classic role and attribute based access scheme, access control list, and two types of token schemes that can be implemented in a stateless distributed system. The first scheme is JavaScript Object Notation (JSON) Web Tokens, followed by macaroons, which seek to address some of the issues with JSON Web Tokens while being easily extensible to accommodate complex access patterns.

6.1 Role and Attribute Based Access Control Schemes

A classic way of organizing access control is to do it based on roles with a role-based access control scheme (RBAC), originating especially from the needs of government and military organizations (Ferraiolo & Kuhn 1992). In RBAC the role or roles of an entity define what resources they can access and how.

Successor of RBAC in access control is Attribute-Based Access Control (ABAC), where authorization decisions are made based on attributes assigned to the requestor, conditions of the environment and the set policy for the target pertaining to those attributes (Hu et al. 2014). In service-to-service context with mTLS, the authorization decision can be based on information embedded in certificates received from an internal CA, as the client certificate is evaluated during the TLS handshake already. This requires the services wishing to leverage this kind of an access control model to coordinate with the CA about the certificate structure and what information is embedded in them. Employing tokens to carry the information is also possible.

A hybrid of the RBAC and ABAC approaches has been applied to microservices in the service mesh solution Istio. It leverages sidecar proxies to provide a generalized solution for mTLS and access control. Services are defined by the service accounts which have a set role. For a REST microservice application Istio allows authorization on a namespace, service, and method level isolation. To allow for more complex access schemes it also supports attribute based constraints, such as IP address or user principal. The authorization decision is a simple allow or deny based on the policy upheld on the sidecar. (Istio 2018)

The challenges with the model are that authorization policies need to be implemented and upheld in every individual service and they need to be able to be updated in a timely manner and pushed automatically to the target services executing the policies. Going to each individual service instance and manually changing policies quickly becomes too cumbersome and prone to error. Depending on the complexity and the size of the system, expressing enough granularity for access control can require a large number of roles which need to be kept updated.

6.2 Access-Control List

Another way of organizing access control is through the use of Access Control Lists (ACL). In this model, the access rights can be represented as a matrix, column per resource (Anderson 2008 Ch. 4.2.2.). Especially popular ACLs are in operating system file security schemes but they have been used in service-to-service contexts as well, for example by the service mesh solution Consul (Consul 2018). Though in Consul the approach is based on tokens tied to policies which mandate the rules for access.

Besides operating systems, ACLs have been applied in data-oriented environments such as the Amazon Simple Storage Service (Amazon 2018). In a service-to-service communication context the ACL implies that the rights are tied into the object which would be at odds with offloading the authorization duties to a sidecar proxy or a dedicated service.

6.3 JSON Web Tokens

The de facto standard of the internet for representing claims has turned out to be the JSON Web Token (JWT) format, as evident by industry support for the standard's track RFC 7519 of the Internet Engineering Task Force (IETF). The principle of a JWT is that it is a JSON object encoded compact way of representing claims that are either cryptographically signed or protected with a MAC. Their compactness requirement is based on the way they are transferred, which is via HTTP Authorization headers or Universal Resource Identifier (URI) query parameters. The usable implementations are JSON Web Signature (JWS) or JSON Web Encryption (JWE) based. (Jones et al. 2015)

A JSON Web Token is formed of three parts, *header, payload, and signature*. The header of a JWT describes the type of the object and what algorithm was used in signing or encrypting it. Payload includes the actual claims, which are statements about the user and additional data that can be decided between the token issuer and consumers. The signature field carries the signature for verifying its legitimacy. (Jones et al. 2015)

Here we create a token "signed" with SHA-256 HMAC scheme using a secret. The payload consists of the official registered claims of *sub, aud, and exp*, which express who the token is for, the intended audience that should accept the token, and the expiration time of the token, defined in the RFC (Jones et al. 2015). In addition, we included the field of *authz* which is our system specific value that tells the target service "Data-Service" declared in the *aud* field what resources they allow the token bearer to access, in this case the resources under "/data/user/123". The target service would then upon receiving a request with this token, verify the signature using the shared secret and check the validity of the claims. If everything checks, the user is allowed to access the particular resources encapsulated and bound by the signature in the token. **Table 6** below represents this example JWT.

Table 6. *Example of a JWT*

Header	Payload

{
"alg": "HS256",
"typ": "JWT"
}
	{
	"sub": "Node-123",
	"aud": "Data-Service;",
	"authz": "data/user/123",
	"exp": 1516239022
	}

The different fields are separated with a dot and the defined token look as follows encoded in base64:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJOb2RlLTEyMyIsIm5hbWUi-OiJFeGFtcGxlU5vZGUiLCJhdXRoeiI6IkRhdGEtU2VydmljZTsxMjMiLCJpYXQi-OjE1MTYyMzkwMjJ9.HxEELVW_k3CXLrOCIF6rydA1UqA-w_563W0b0rCdODE

6.3.1 Security of JSON Web Tokens

In the RFC 7519 defining JWTs, the HMAC with SHA-256 based *HS256* and *none* are the algorithms that are mandatory to implement as outlined in chapter 8, "Implementation Requirements" (Jones et al. 2015) but several other HMAC and public-key algorithms can optionally be implemented. Let's look at take the algorithms used for implementation

comparisons at JWT.io (JWT 2018), the options provided are (all of them use SHA2 hashing):

ECDSA – *ES256, ES384, ES512*

HMAC – *HS256, HS384, HS512*

RSA – *RS256, RS384, RS512*

Elliptic-Curve based algorithms have their security of their key length divided by two bits of security (Barker 2016). For the ECDSA options it means 128 bits, 192 bits, and 256 bits respectively. This is achieved when using at least SHA256 or stronger hashing function. All of the options are thus appropriately long for current and future use according to the NIST and ECRYPT standards (NIST 2013; Smart 2018). Though, some concern has been expressed over using ECDSA due to the weak provable security guarantees of the scheme (Smart 2018).

The HMAC constructs are also considered secure, corresponding to 128 bits, 192 bits and 256 bits of security respectively for HS256, HS384, and HS512 (NIST 2015; Smart 2018). For the RSA options to match the 128 bits of security provided by SHA256, the RSA key length needs to be 3072 bits or longer (Barker 2016). This means that a commonly used key size of 2048 bits would not be sufficient for future use as it offers only roughly 112 bits of security (Smart 2018).

In general, the HMAC option has the least concerns over security. It enjoys significantly faster operations than public-key solutions (Spomky labs 2018) and less implementation complexity. The only downside is that verification and token minting have to happen with the same key, expanding the threat surface for key exposure. Using HMAC also does not provide non-repudiation, but this is not a particular problem in this use case. If the system requirement is that keys not be shared and asymmetric cryptography should be used, the RSA options offer faster verification with slow signing operations as made evident in the earlier comparisons in section 2 chapter 4.

Allowing for several algorithm options has proven to be an implementation pitfall when the user is allowed to choose the algorithm without proper validation. This has led to security vulnerabilities in popular JWT libraries. When we wish to verify a token was minted by an appropriate party, we have to first look at the *alg* field in the header to determine which algorithm we use for verification. As we have not validated the token we have not validated the header either and we give the possible adversary the power to select the algorithm used for verification (McLean 2015).

In the naïve case the power given to the adversary to choose the algorithm for verification with the *alg* parameter in the token header can be abused by setting the algorithm to *none* which is mandatory to implement as per RFC 7519 (Jones et al. 2015). With this, the

attacker could supply an empty signature and the target service allow it (McLean 2015). Thus, unless there is an actual use case for the *none* algorithm to be used, any token with that value in the header should be outright denied. This is one of the implementation concerns that the developer implementing a library to handle JWTs has to deal with. The scheme is not necessarily secure by default.

We are still left with the adversary controlling the algorithm field. Another vulnerability was found in the library *jwt-simple* versions 0.3.0 and earlier in 2016 and reported to the National Vulnerability Database (NVD) maintained by NIST (NVD 2016). The library is a popular JWT encoder and decoder for NodeJS (over 70 000 downloads from NPM per week as of 8.10.2018) (NPM 2018). In the jwt.decode() function the library implements, it was possible for a malicious user to choose the algorithm used for verification (NVF 2016). If the server was expecting the use of an RSA algorithm but instead the adversary sent a header with HMAC-SHA with the RSA's public key, the server treated the public key as the private key of the HMAC (McLean 2015). As the public key is meant to be shareable it was trivial for the attacker to gain access to it. This allowed the adversary to forge tokens with arbitrary data and pass the verification (NVD 2016). The vulnerability can be countered with enforcing the use of key identifiers per algorithm.

Other than the implementation pitfalls, the security of JSON Web Tokens is dependent on the underlying cryptographic protocol and the primitives the protocol is comprised of. For implementations, trusted and vetted open source libraries are usually the best bet, unless there is justification for custom implementation. JWTs enjoy a great deal of industry support as can be seen from the amount of languages with library support (JWT 2018) and their usage in established standards such as in the authorization framework OAuth 2.0 (Hardt 2012).

6.4 Macaroons – Cookies but Tastier

In the paper "Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud" Birgisson et al. (2014) outline a new scheme for authorization credentials especially designed for decentralized cloud environments. In general, they are bearer credentials that in their most basic function allow for access control in the manner of "if you are in the possession of this credential you are good to go". But unlike normal web cookies that do not have any sort of identity attached to them in stateless manner or allow for derivation of new credentials, macaroons promise attenuation and contextual confinement that allow for more fine-grained access control (Birgisson et al. 2014). This makes macaroons a great fit for REST microservice architectures on paper.

Macaroons promise to allow services to delegate access rights with restrictions to their access scope and allowed context (Birgisson et al. 2014). An entity with full access rights can for example derive bearer credentials from their credentials that only allow the other entity to access a certain file from a certain authorized device. The original research paper

presents macaroons mainly in user-to-service context but we will explore their suitability to service-to-service use. The question that remains is can the advantages of macaroons be leveraged effectively in an inter-service communications context to justify choosing them for authorization.

6.4.1 Macaroon Construction

On a high-level, macaroons are constructed by producing an HMAC tag of the token content to provide message integrity and authenticity (Birgisson et al. 2014). Due to the nature of HMAC, macaroons can be created in layers by adding more and more caveats, each time producing a new signature of the token content. They are then verified by re-constructing the chain of HMACs and comparing the resulting keyed digests.

First a nonce is needed. Nonce in this context means an arbitrary cryptographically secure random or pseudo-random number (Boneh & Shoup 2017 p. 80). This serves as the opaque key-identifier that maps to a secret root key only known by the service (Birgisson et al. 2014).

1. Using this nonce we create our first macaroon without any predicate caveats. It allows you to access everything in the target service. We do this by HMAC(root key, nonce) -> we get the signature of this macaroon. Let the signature be ***ab..35****.*

2. Next we want to create a bearer credential that allows another service to access the storage service, but only the blob 101. We now take the macaroon created in the phase 1 and add the predicate caveat blob = 101.

Signature is created by: HMAC(ab..35, "blob = 101") = ***cf..23***

3. Several predicate caveats can be added in one go. We want this macaroon to grant access to the blob 101 but only if the organizational unit in their certificate is 'data' and if the time passed since the epoch is less than 1540218243.

The macaroons created in these three steps are demonstrated below in **table 7**.

Table 7. *Example Construction of a Macaroon*

Macaroon 1	**Macaroon 2**	**Macaroon 3**
NONCE ***ab..35***	*NONCE* *Blob = 101* ***Cf..23***	*NONCE* *Blob = 101* *OU = Data* *Time < 1540218243* ***UF..52***

All the aforementioned caveats are considered first party caveats as they do not require any additional verification or authentication from other sources. The biggest edge of macaroons over JWTs lie in the third-party caveats they make possible. For example, in the last macaroon we could augment it with a third-party caveat: “present a valid token from the authorization service X” which the target service verifying the macaroon would then be able to verify from the external authorization service X.

6.4.2 The Security of Macaroons

Macaroon’s security is (bar poor implementation) based on the security properties of the used HMAC algorithm (Birgisson et al. 2014), which in turn means the construction’s security is reliant on the underlying one-way hash function and the used key (Krawczyk et al. 1997; Bellare et al. 1996). This means that with a SHA2 hash-function with hash values longer than 256 bits or SHA3 we have a construct that has not been proven insecure as per the requirements of NIST (NIST 2015) or ECRYPT (Smart 2018).

As macaroons use HMAC as a black box, a macaroon solution can be made more secure by choosing an HMAC algorithm with stronger security properties, such as a SHA512 instead of SHA256. This will increase the size of the signature which affects for example the amount of bandwidth used and storage requirements. Performance wise it is not as clear as that. On a 64-bit system SHA512 outperforms SHA256 as it operates on longer words (64 bits vs. 32 bits) (NIST 2015). This can be seen in the Libre SSL 2.2.7 speed benchmark run on Intel Core i5 2.3 GHz on MacBook Pro 2017. The results of which are presented below in **figure 4**. The algorithm choice thus depends on the use case and what resources to optimize for.

```
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
sha256            8940.65k    18837.66k    30654.21k    36904.75k    39266.80k
sha512           17313.97k    68557.59k   103179.47k   147933.41k   170031.95k
```

Figure 4 SHA256 vs. SHA512 performance

6.4.3 Challenges of Macaroons

Macaroons sound pretty great at this point. They offer an arguably more extensible alternative to JSON Web Tokens (JWT), while avoiding an implementation pitfall of JWTs by only allowing the use of HMACs (Birgisson et al. 2014) which removes the risk of abusing mismatched algorithm parameters. Macaroons do indeed have their own problems as well. One we quickly run into with caveats is the issue of logic formalization. To ensure reliable contracts and interoperability, the logic needs to be standardized throughout the interacting services. As by design macaroons are lenient and non-descriptive as how to implement caveat logic. This puts pressure on developers to produce internally

consistent logic and good documentation of the logic to ensure inter-service compatibility. This has perhaps been one of the biggest challenges in macaroons adoption in the larger industry. Now, four years after the original paper was published, macaroons still remain a curiosity to the greater industry compared to the popularity of JWTs.

For practical purposes, in service-to-service context most of the finer points of third party caveats end up being underutilized. Compared to JSON Web Tokens these caveats are one of the big differentiators for macaroons, but in practice it is quite hard to justify the engineering effort required to develop standardized libraries and formalize caveat logic. Especially for the third-party caveats which require interoperability from the third parties. This becomes cumbersome in the microservices world where generalizing as many utilities as possible and focusing on the difference making application logic is sought after. The best use case for the third-party caveats seems to be in user-to-service contexts where more performance overhead and latency is tolerated in order to allow for more complex access schemes as an alternative to authorization schemes such as OAuth 2.0 using JWTs (Hardt 2012).

The verification method of macaroons also sets up an engineering challenge in managing secrets. As the minting and reconstruction of a macaroon requires the root secret key, this inevitably means one of two things. Either the minting and verifying is done solely by that one service or the secret needs to be shared, which in turn requires methods for secure distribution and management of secrets. The pros and cons are gathered below in **table 8**.

Table 8. *Advantages and Disadvantages of Macaroons*

Pros	Cons
Allows only a secure HMAC option	Formalization of logic needed
Third-party caveats enable novel options for authentication and authorization	Lack of industry support
Security reliant on underlying one way hash function used as a black box	Minting and verification using symmetric cryptography creates secrets management challenges
Attenuation and delegation of access allows for more flexibility	Lack of standard implementation leaves a lot of responsibility for developer
Enables granular resource level access control based on set authorization policy	The more you want to do with them, the more logic needs to be implemented on application level

6.5 Authorization Models

In this section, we will apply the learnings from studying different authorization schemes to produce authorization models that could be applied to real-world systems. As we move on to discuss authorization logic, we make the assumption that the underlying communication channel offers confidentiality, message integrity, replay protection, and origin authentication. We look at two different authorization models using bearer tokens and a central authorization service and one model where the authorization decisions are done by a per-service proxy. The **table 9** below gathers the notation we use for our protocol descriptions in this chapter.

Table 9. *Protocol Notation Used*

Notation	**Meaning**
A, B	Principals
AS	Authorization Service
$Gen_{Mac}(K, cav)$	Mints a macaroon with the caveats *cav* using the key *K* to produce an HMAC tag
$Att_{Mac}(M, cav)$	Creates a macaroon with the added caveats *cav* to the macaroon *M*, chaining the caveats with an HMAC signature
$V_K(M)$	Verify the legitimacy of the macaroon given as input *M* using the key *K*
$Gen_{Token}(claims, A, B)$	Creates a JWT token with the set of claims and the requesting principal and target service
$S_{ski}(T_n)$	Signs the token T_n with the secret key of *i*
$V_{pki}(T_n)$	Verifies the signature of token T_n using the public key of *i*
$A \rightarrow B: i$	Send a message from A to B containing *i*
$x \leftarrow y$	Assign value *y* to *x*

In these models, we are concerned solely with the interaction required for a service to be granted access to the desired resources as per the defined policy. Principal names are added in the protocol messages to adhere to the prudent engineering practice of "If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message" as described by Abadi and Needham (1994). This name does not necessarily mean a name as we humans commonly understand but a cryptographic identity given to a particular service instance. What this achieves is that would a bearer token be leaked, it would not be usable by any other entity than the

one it was created for in the target service bound by the signature or authentication tag. This can be described in more detail in the token constraints in system specific terms.

6.5.1 Authorization Service as a Token Minting Proxy

In the token proxy model, we consider a system where authorization is handled together by the target services and a dedicated authorization service. This example is provided using macaroons to enable an authorization model that would not be practically possible with JWTs. We utilize macaroon based tokens to construct a system where we can reap the benefits of fast HMAC token verification and the extensibility, delegation, and attenuation of macaroons, while keeping the authorization policy centrally maintained for operational convenience. The extra cost is the interaction required from both the authorization service and the target service in order to produce requesting service an access token. The simple representation of the model is presented below in the **figure 5**, followed by a more in-depth representation of the scheme in protocol notation.

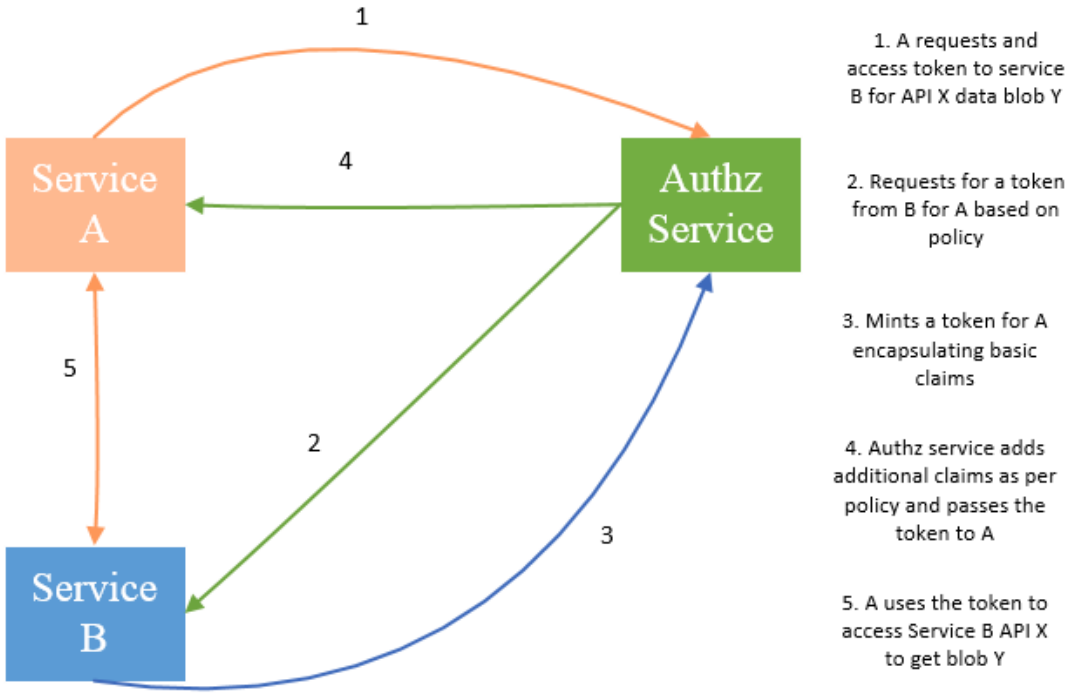

Figure 5 The Token Proxy Model with Macaroons

(1) A → AS: (A, B)

(2) AS: Eval(P, A, B)

(3) AS: $m_1 \leftarrow$ (A, B)

(4) AS → B: m_1

(5) B: $\text{Token}_{BAS} \leftarrow \text{Gen}_{Mac}$ (K, caveats, A, B)

(6) B → AS: $Token_{BAS}$

(7) AS: $T_{AB} \leftarrow Att_{Mac}(Token_{BAS,}$ caveats)

(8) AS → A: T_{AB}

(9) A: $m_2 \leftarrow$ (request + T_{AB})

(10) A → B: m_2

(11) B: $V_K(T_{AB})$

In this example scenario, Service A wants to access an API X of Service B.

1. *Service A requests a token from Authorization Service (AS) to access resources of Service B.*
2. *Authorization Service evaluates the policy on whether A should be able to access B.*
3. *AS creates a request for token for A to access B, including A's identifiers from a certificate or using other strong identifiers.*
4. *AS sends the request to B.*
5. *B mints a token using its secret key, a set of caveats, and the identity of A and target B.*
6. *B sends the minted token back to AS.*
7. *Based on the earlier policy evaluation AS adds the access A is granted as caveats in the macaroon T_{AB} using HMAC chaining without having to know the secret key.*
8. *AS sends the freshly minted token to A.*
9. *A adds the received token as part of the request to B.*
10. *A sends the request to B.*
11. *B verifies the macaroon by reconstructing the HMAC chain using its secret key.*

The caveats target Service B mints a token with can be things such as expiration, audience, and other basic generic claims. Authorization Service then based on the policy it upholds decides what access the requestor needs and restricts the macaroon to appropriate access rights with more detailed caveats and chaining them to the pre-existing ones. In this case even if the authorization service was compromised it would not be able to use this token as it has the identity of A bound to it with an authentication tag. A can then use the token to access the required resources at Service B for the lifetime of the token.

Trying to realize this model with JWTs runs into the problem of losing on delegation and attenuation, meaning that the token needs be minted by the authorization service directly, in which case relaying it to the target Service B makes little sense and another architectural choice is more suitable. In the case where the target Service B mints the token for the requestor, the Authorization Service reduces into a simple proxy due to being unable to add caveats and the policies still need to be implemented in B or the relevant policy section needs to be passed onto the target service from the authorization service if we

want to hold on to centrally managed policies. This introduces complexity and reliance on reliable and timely passing of the policy to the executing services.

For minting tokens with an HMAC construct, it requires that the minter and the verifying service share a secret key. In this case as the original token is minted by the target service and also verified only by it, the secret does not need to be shared anywhere else. The AS is able to add caveats to the token and attenuate it without being privy to the secret minting key. If the Authorization Service mints the original token, the secret needs to be shared at minimum two locations and robust secret management methods are needed. An important factor is the bootstrapping of the service, where on creation the service needs to know the secret to be able to verify any tokens. In order to have an efficient and secure solution, there need to be mechanisms for secure secret distribution.

Another option is to use a private-public key pair based solution for JWTs but the token proxy approach starts to make less and less sense in that case. The target service would create the token with its private key but then the verification would still be done on the target service using the public key. This leads to a situation where we have opted to use a slower crypto algorithm without adding any meaningful security. On the other hand, the authorization service can create the token with a private key corresponding to a public key the target service has, which does not describe the token proxy model anymore and will be considered in the next model.

The revocation mechanism of the token is mainly based on its lifetime, shorter validity limiting the effects of a leaked token but requiring more frequent fetching of a new token. Another option is to rotate the secret key used to mint and verify the tokens. This would then naturally revoke all the tokens minted with that key and trigger a re-fetching of tokens for all the services needing access to the target service. In the case a compromised service is observed, the policy on the authorization service can be changed to not grant anymore tokens to the compromised service, at which point the duration of the possible exploitation of the compromise is limited to the validity time of the token. At that point, even when the certificate of the compromised service is valid, it will not be able to access any resources of other services without a valid token.

Another option for revocation would be achieved by using different keys per requesting service or a group of services. Deleting or rotating that particular key would offer instant revocation but at the cost of increased risk of key exposure and requiring robust key management and storage which introduce more system complexity. Also, the idea of long-lived refresh tokens as a basis for fetching new ones could be used. But in a system with certificates and mTLS the same effect can be had by basing the authorization policy on a cryptographically strong identifier embedded in the certificate. On compromise the policy would be changed on the central policy source and again the exploitability of the existing token is limited by its validity.

With using a central authorization service the downsides are in being able to scale the service and resiliency. The AS going down means putting a hamper on the system as new tokens cannot be provided. Resiliency can be increased by replication of the service, which requires being able to fetch an updated policy at the time of the AS being spun up. As the authorization service does not need knowledge of the key to add caveats this reduces the required key management and makes replication easier.

6.5.2 Authorization Service as the Token Issuer

As with the previous model proposed, we set to identify an authorization model with the assumption that establishing strong identities and authentication are solved. All the traffic between various services uses mTLS and the communication channel has the guarantees of confidentiality, integrity, authenticity, and replay protection.

The flow of authorization is quite straightforward in this model as the access control policy enforcement and access token minting is done by a single central authorization service. This simplifies the architecture as we have a central source for access policy enforcement and to create the policy implementing bearer tokens. In the example Service A wants to access API X of Service B. Both A and B are microservices part of the same application. In the **figure 6** we have a simplified picture of a requesting service fetching a token and using it to access a target service resource. It is followed by a description of the scheme in security protocol notation.

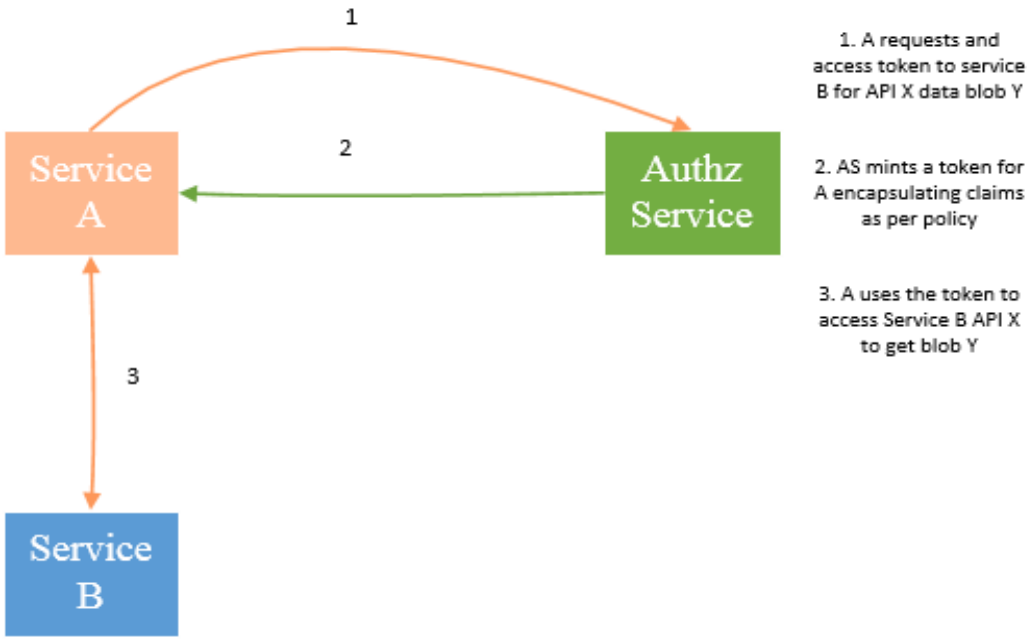

Figure 6 Authorization Service as Token Creating Central Policy Source

(1) A → AS: (A, B)

(2) AS: Eval(P, A, B)

(3) AS: $Token_{AB} \leftarrow Gen_{Token}$ (claims, A, B)

(4) AS: $T_{AB} \leftarrow S_{skAS}$ ($Token_{AB}$)

(5) AS → A: T_{AB}

(6) A: $m_1 \leftarrow$ (request + T_{AB})

(7) A → B: m_1

(8) B: V_{pkAS} (T_{AB})

1. *Service A requests Authorization Service (AS) to grant a token to access Service B.*
2. *AS authenticates the requestor and makes a decision based on the policy it has for the target service whether to mint a token or not.*
3. *If the policy states that Service A can access the requested API X, Authorization Service mints an access token to access that API*
4. *AS signs the token with its private key to bind the claims to this signature and prevent tampering with it. Here the key used could also be a symmetric key shared between AS and B or a private key explicitly used for only B.*
5. *Authorization Service passes the token to A*
6. *A attaches the received token as a part of the request to B*
7. *A sends the resource access request to B*
8. *B verifies the signature on the token, parses and evaluates the claims in the token and makes the decision whether the claims match A's request.*

In this model, both JWTs and Macaroons can be used. We do not consider any complex access schemes where third party caveats would come in use, but a simple model enabling granular to control access to resources. The requirement set for this system is that the minted token can be verified by the target service. In the actual minting of the tokens we have asymmetric algorithm options with JWTs, and HMAC options provided by both macaroons and JWTs (Jones et al. 2015; Birgisson et al. 2014). The choice whether JWTs or Macaroons are used is based on the system requirements. In a very straightforward access case such as this one, JWTs provide the more attractive option as a lot of standardized libraries are available for variety of languages and they provide the flexibility of choosing between symmetric and asymmetric options.

Let's consider the asymmetric signing option first. The authorization service mints the token with the private key corresponding to the target service and the target service can then verify the minting with the public key it has corresponding to that particular private key. Doing it the opposite way goes against the principle of asymmetric cryptography as

public keys are designed to not be secret and to be shareable (Menezes et. al. 1997 p. 283). If the public key used for minting in this case was to be revealed it could be used to mint tokens by whoever and they would pass verification as the key they were minted corresponds to the private key used for verification. The control over who is able to mint tokens would be lost.

The second option is HMAC based minting where the token is verified with the same secret as it is minted with. In essence, we recreate the token at target service and see whether the signatures of the token we received and the one we created match. The performance edge of HMAC scheme comes with the caveat of the secret needing to be shared with the token generating entity as well as the verifying entity. This puts pressure on good secret handling and management of securely storing, transporting, and including them into the services when the minting is not done solely by the target service. Compared to an asymmetric solution where the secret only needs to be stored in the minting service, we are talking about the secret being stored in at least twice as many places. Secret being used in several places means a bigger attack surface and more chances for the exposure of that, as well as requiring monitoring for credential compromise. This is especially true in large scale microservice world where individual microservices are constantly being spun up and spun down depending on the scaling needs. This would set a requirement for a robust way of generating, distributing, and rotating secrets.

Another option instead of the target service verifying the token itself is to pass it back to the authorization service for verification of the token. This would also essentially double the number of requests needed to fulfill a request as per *n* client requests, the target would also need to make *n* requests to the authorization service to verify the tokens. For high performance services this represents an unsustainable coupling of the services to the authorization service.

As in the previously discussed model, the validity period of the token defines the possible window of exploitation of a leaked token. Further adversarial effects on other services can be clamped down by changing the token granting policy. The policy change does not negate the token already created and the service will still continue to have access with that token unless a key rotation is done. Embedding usage restricting claims about the requestor and audience into the token limits the effect of compromise.

6.5.3 Services Upholding Their Own Authorization Policy

In the two previous models, we saw the authorization policy being upheld by the authorization service and the target services verifying the legitimacy of the token and acting according to the information in the bearer token. Another way to organize authorization in a microservices architecture is to let the target services themselves or proxies in front of them handle the authorization. There are a few ways to implement this. The simplest

one is to maintain a per-service authorization policy on the service itself and make decisions based on the information in the certificate services use to authenticate themselves. The policy then maintains what that particular entity is allowed to access.

The policy can be for example in the form of ABAC (Hu et al. 2014) or RBAC (Ferraiolo & Kuhn 1992). The choice of policy type mandates what information must be embedded in the certificates in order to be able to make decisions based on it. For example, Istio authorization is handled with RBAC as the default option (Istio Security 2018). In this model, all the requests coming in to the sidecar proxy are evaluated against the implemented policy passed down from a central policy source (Istio Security 2018). The sidecar then makes the decision of DENY or ALLOW based on the requestor and the configured policy of who is allowed to access what. From a network load perspective in a sidecar model this means that an extra hop to the authorization service is omitted.

Another option is to leverage tokens similar to the central authorization models, difference here is that each service grants and verifies their own tokens and an authorization service can be omitted but requiring policy management on the service itself. This approach has the benefits of each service being able to define how they want their tokens to be structured without having to care about interoperability with other services. From a performance point of view, it is easy to use the more performant HMAC based algorithms in this case as the symmetric key does not need to be shared with other services. On the other hand, more customization means less reusability and more total engineering effort.

Let's look at the two options and what they would look like.

Option 1, RBAC, ABAC, or ACL:

(1) A → B: (request, A, B)

(2) B: Eval(P, A)

1. A sends the request of accessing the particular resource on B through its sidecar.

2. B's sidecar evaluates the request based on the policy it upholds. Decision is either ***ALLOW*** *or* ***DENY.***

The option is very simple, the requestor's access is evaluated based on the policy B upholds. In the case of role or attribute-based access control, the basis for decision making can be for example baked in the certificates each of the services has from the internal CA and which are presented during the mTLS handshake. **Figure 7** below represents the simple non-token authorization model with sidecars. To make the solution more scalable and allow central policy management, we introduce a central policy source where the services fetch their policies.

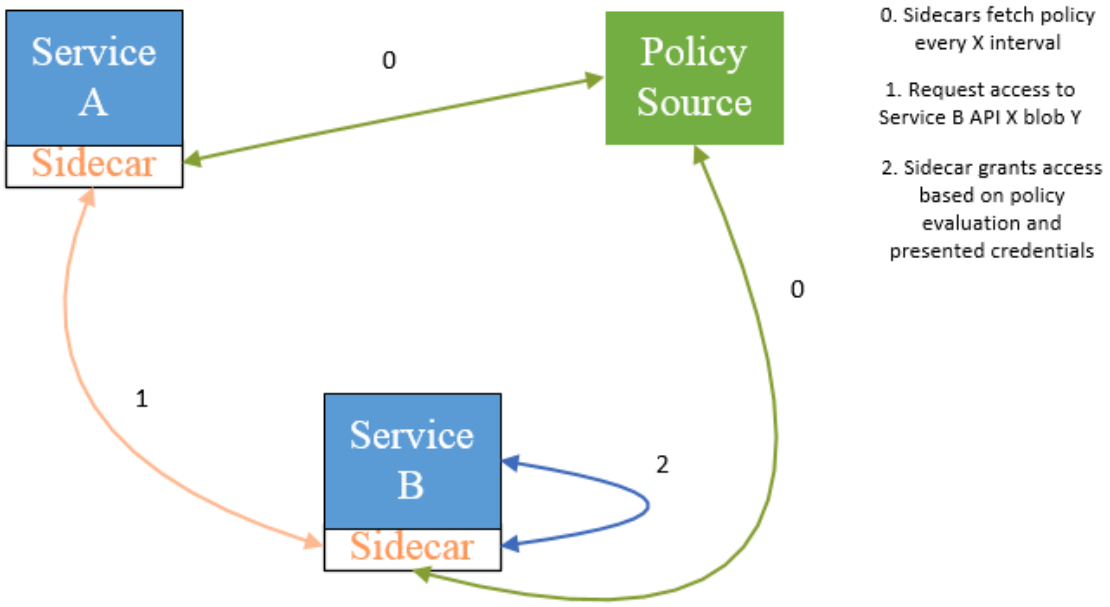

Figure 7 Authorization with Sidecars

Option 2, bearer tokens with HMAC option:

We have opted to present the HMAC option here as the token is minted and verified by the same service and the token should not be used anywhere else. Either JWTs or macaroons can be used.

(1) A → B: (resource, A, B)

(2) B: Eval(P, A)

(3) B: $T_{AB} \leftarrow Gen_{Token}$ (K, claims, A)

(4) B → A: T_{AB}

(5) $m_1 \leftarrow$ (request + T_{AB})

(6) A → B: (m_1)

(7) B: $V_K(T_{AB})$

1. A sends the request of accessing the particular resource on B through its sidecar.

2. B's sidecar evaluates the request according to the policy P it has for A.

3. B creates a token, binding the access to the requested resources with the secret key K used in HMAC

4. B sends the created token to A

5. A adds the token to the requests

6. Request is sent to B.

7. B's sidecar verifies the legitimacy of the token with the secret key K and grants access to A

The security implications of both options are the same as the decision for token minting and policy based authorization are based on the cryptographically strong identity represented by the certificate and the claims embedded in it. As long as the minted token has at least the same or stronger strength than the certificate, the security baseline is set by the strength of the certificate. The security then boils down to the trust on the internal CA and the certificates it signs. The performance of a token based solution is reliant on the verification operation and parsing the token, while as for the ABAC or RBAC solution the performance is reliant on policy parsing and evaluation. Support for tokens in service-to-service context might be mandatory later if external requests are handled and principal propagation is needed.

To alleviate the pain of managing per-service policies manually, the policies can be maintained in a central policy service which takes care of distributing the policies to the appropriate services when updated, allowing central policy management with distributed policy enforcement. This is especially effective in a sidecar proxy based solution where access control decisions are handled by the sidecar based on generic policy template customized to fit per-service needs. This has the added benefit of being able to store policy configurations in version control systems and to keep track of changes. This approach has been leveraged in industry for example in the Istio service mesh with RBAC (Istio 2018). An extra step of security that can be implemented with this active policy distribution mechanism is making policies expire and require them to be refreshed at appropriate intervals to reduce the window of possibly acting on stale information. The better synced the policy store and the service are, the smaller this window is.

Performance implications of which solutions is the fastest are very hard to gauge. For Istio v. 0.71 (current version at the time of writing being 1.0) an estimate of 10ms of overhead per service-to-service call is given, using their authentication infrastructure for a sample Java application, including the request going through originating and target sidecar proxy, telemetry collection, and a policy check (Istio Performance 2018). In the test setting presented by Yarygina & Bagge (2018) they arrived at an estimation of 7% decreased performance from the use of JWTs. Though the researches do not give any details regarding the requests, which makes it hard to analyze the general applicability of the results.

The non-token based authorization models saves one request that is used to request a minted token which in the grand scheme of things is a negligible cost, if that token can be used from tens to thousands or more requests. In general, the heavier the computation associated with the request, the less overhead the authorization portion represents. In these cases, the heavy optimization of very low overhead security computation is less

important and the approach offering more bang for the buck is probably to go for optimization of other areas. In an environment where individual requests are small and constant, the overhead is definitely worth considering.

6.6 Principal Propagation in Authorization Models

We presented the confused deputy problem in the beginning of this thesis, where a service fulfills a request on another entity's behalf and the problems this might cause. Even if we do not consider end-user to service communication, a situation like this can arise in a service-to-service context if for example, we have a data aggregation services that fetch data on behalf of the client and operate on it before delivering it to the requestor. In these cases, we need to identify the identity of the original requestor and whether they are allowed to execute the requested actions as the intermediary service here is not necessarily privy to the authorization policy of the target service.

With the bearer token, central authorization service based systems, the service making the call to a further service can attach the requestor's token in the request, if no audience is specified in the token or the target service has defined in its policy that the aggregating services are allowed to make requests like this. In a simple model, the token encapsulates the identifiers of the requestor and the ultimate target service trusts the calling service to authenticate the requestor. The secondary target service then authenticates the caller service when establishing a mTLS connection. After, the target service makes the authorization decision on the requested resources based on the verification of the received token and the information in it based on the original requestor.

The requirement is that either all the services in the chain have a shared key for verification, or the intermediary service encapsulates requestor identifiers in a token that it signs, which can be then verified by the target service using either a corresponding public key or a shared symmetric key, depending on the implementation. On the other hand, this opens up an avenue for a compromised intermediary service to either harvest tokens or craft malicious request to the target service using spoofed requestor identifiers.

Alternatively, the token can be used for only the first hop and then the service propagating the principle requests for a new token to be used at the next service, encapsulating the requestor's identifiers as well as their own (Posta 2018). The downside is the increased number of calls and tokens needed. In either case, infrastructure is needed to effectively distribute the needed keys to the correct services, for example by leveraging the orchestration solution along a secrets management system. The least complex solution is to define in policy that the aggregating services have access to a larger amount of services and data. The gain on simplicity comes with the cost of compromise increasing.

For external parties, access and propagation is made possible through an API gateway that identifies them based on e.g. credentials, certificate, or an API key and then requests

for a token from an authorization service or mints one itself to represent the requesting entity and its request inside the system. The internal services then need to possess the corresponding public key to the private key used to create the token in the first place or in the case of symmetric key signing, the token minting key needs to be shared for verification. Solution similar to this was proposed in the Missfire model by Yarygina & Bagge (2018) with leveraging a security token service. For example, OAuth 2.0 as described in the RFC 6749 of the Internet Engineering Task Force (Hardt 2012) can be used for authorizing external service API access.

7. ENGINEERING A SYSTEM TOGETHER

In the beginning, we established some standards for a new system for better microservices security. We have gone over the security requirements and defense-in-depth mechanisms that can be implemented on the upper three levels of the container cloud deployment stack, meaning communications, application level, and orchestration. In this section, we will examine how these identified mechanisms can be pieced together into a cohesive distributed system and what are the concerns we have to take into account when discussing the high-level system design choices. Architectural choices are particularly important as they are among the first decisions made in software development lifecycle and their impact carries throughout the whole lifecycle. We propose two distinct ways to organize a microservice architecture with proper defense-in-depth protections against the defined adversary inside our network.

We will evaluate the proposed architectural structures based on requirements set for the system. The requirements can be divided into two types. Functional requirements describe behaviors the system needs to implement. Quality attributes on the other hand are system criteria it can be judged based on. This methodology presented here is a very limited scope version of the full requirements engineering scheme. The functional requirements are based directly on countering the defined adversary and limiting the effect of compromise with these defense-in-depth methods.

In our analysis, we use the Architecture tradeoff analysis method (ATAM) as the base to construct our analysis on. Due to the structure of ATAM as describing the architecture process from design to implementation, we skip parts of it and focus on the steps four and six of the methodology, namely identifying the architectural approaches and analyzing the identified architectural approaches (Kazman et al. 2000). This architectural analysis used is known as the software architecture evaluation (Bass et al. 2012).

Functional requirements that guarantee the desired security properties for communications are covered by the use of mTLS as long as the internal PKI can be trusted. The system also needs to be able to implement granular, resource level access control based on an authorization policy. The ways to achieve these goals were explored in the previous sections on authentication and authorization. The last functional requirement is that the system can be automatically bootstrapped. This means that the deployment sequence of a fully functional system can be defined as code and automated to run. This is noted as one of the prerequisites to enable frictionless delivery from development to production, which was identified as a key to success with cloud-based solutions by Kang et al. (2016). Security-wise it means that all the security properties can be bootstrapped as well and systems work in a secure way by default when spun up. The caveat to this requirement is the increased security engineering effort that ensues. For example, solutions for robust

secrets and certificate management need to be implemented in order to be done automatically.

The chosen quality attributes are by necessity generic and lack concrete metrics as the models do not directly represent any real-life system. The attribute of minimal performance overhead is tied to minimizing operational costs through reducing cloud resource usage and minimizing redundancy. Slower loading times for the end-user also have an adverse effect on their interest, for example, based on a dataset of 10 billion user visits in online retailing, Akamai found that only a 100-millisecond delay in website loading time can mean a 7 percent reduction in how often a visitor went from browsing to taking the retailer desired action (Akamai 2017).

The horizontal load scalability requirement is based on the goals of microservice design and containerization bringing per-service scalability by replication (Guerrero 2018). The added security features should not interfere with this goal more than necessary. Platform agnostic solution is based on the multi-cloud deployment requirement to ensure better performance in the business environment where seeking extremely low response times mandate global service distribution to serve a global audience.

The attribute of requiring minimal code changes is based on the empirical observation that for any solution, the less effort required for implementation, the more likely it is to be embraced. Especially true this strikes in security where demonstrating the value of added defense methods are hard to quantify and even the developed methods such as the stochastic modelling approach by Madan et al. (2004) are difficult to implement to evaluate a real-life system accurately. Whereas the implicit value of security that springs to the minds of security engineers is often not as clear to engineers in other disciplines. **Table 10** gathers the chosen requirements for the system.

Table 10. *System Evaluation Requirements*

Functional	Quality Attributes
Service-to-service communications are encrypted providing confidentiality	Minimal performance overhead caused by security
Service-to-Service calls are authenticated and provide message integrity	Horizontal load scalability
System implements granular, resource level access control	Implementation requires minimal source code changes
Infrastructure and deployment can be defined as code and automated	Platform agnostic

The two models, proxy and service mesh, presented next, were chosen from among the patterns described before. The models adhere to the microservice principles of low decoupling and offer great potential for scalability of services. They are presented in their basic and generic form; further management layers and abstractions would be needed in actual large-scale deployments.

For both services, we assume a generic secure cloud environment in which there exists a trusted internal PKI providing certificates for legitimate entities. They piece together the areas introduced in the previous sections, they leverage container orchestration to secure service hosting, establish mTLS connections through sidecars or application code, and have a granular authorization scheme. The sequence from spinning up a new service to it accessing resources from another service is generally as follows:

1. *The orchestration spins up the first manager node that will point to an existing internal CA.*

2. *A cluster of containers is spun up on the nodes. The certificate of the new CA is distributed to the nodes and they get a certificate from the CA. Service instances are deployed on the cluster.*

3. *When nodes in the cluster establish communications between each other, they mutually authenticate based on the certificates they established from the cluster's CA.*

4. *When a service instance wants to communicate with an instance of another service in the same network, they mutually authenticate each other based on the certificates they have established with the same internal CA and establish a secure communication channel over HTTPS.*

5. *To access resources on another service, they go through the authorization scheme to access the service, either getting a bearer token or using RBAC/ABAC.*

7.1 Proxy Mesh with Central Authorization Service

For the first model, we look at the previously introduced proxy mesh model with a central authorization service minting tokens and upholding authorization policy. The advantages of this system are the non-reliance on sidecar containers alongside application containers which means reduced deployment complexity with less moving parts. Using a centralized authorization service instead of per-service methods, means that the configuration and maintenance of policies can be done in one place and the changes are reflected instantly. On the other hand, the authorization service introduces some coupling effect of services as they are reliant on it to provide tokens for resource access. The authorization service can be scaled independently with replication of instances to improve performance and reliability (Jhawar et al. 2013).

We leverage the container orchestration in providing us with the baseline for getting certificates and bootstrapping mTLS connections inside the cluster. Then per service instance certificates are used for mTLS connections between services. We rely on native implementations of TLS instead of a sidecar container.

The role of the inter-service proxy mesh is to offload service discovery, load balancing, and other networking functions from the service to the proxies (Stetson 2018). In a defense-in-depth model, TLS connection termination and establishing cannot be solely left for the proxy mesh but needs to be also implemented on the service-to-proxy connections. On the other hand, the services need to only be able to verify the certificate chain of the proxy mesh that should be based on the internal CA. The challenge for the services is parsing the received certificate and authenticating based on that.

The proxies handle connections coming in through a possible API gateway from the clients or other external sources and pass the requests to the appropriate service with identity propagation. This frees the services to not have to worry about being able to verify the certificates of external services or having a non-internal CA granted certificate, as the TLS can be stripped on proxy and re-encrypted for the target service. From a business perspective, this means less operational costs associated with certificates.

When TLS connections are established between all the communicating entities and requests are only passed using this secure channel, all the messaging is guaranteed message authentication, replay protection, integrity, and confidentiality. Additionally, with mutually authenticated TLS, the entities can be certain of who they are communicating with based on the certificates and strong identities based on the certificates signed by trusted internal CA. This satisfies the goals of message integrity, confidentiality, and authentication stated in the functional system requirements.

As there is no generic solution shared between all the services where changes can be made uniformly across the system, standardized interface contracts become vital as also highlighted in the REST design principles (Costa et al. 2012). Opting for the native code solution allows for a lot of customization but also requires proportional engineering efforts. The model can be supplemented by a control layer such as the Docker Enterprise platform to get a higher-level view into the system, or the control layer can be a custom solution pieced together from pieces like separate monitoring and logging services. The platform combined with orchestration bootstrapping mechanisms aims to satisfy the functional requirement of automated deployment defined in code.

The choice of bearer token in this model is JSON Web Tokens (Jones et al. 2015), due to their wide adaptation, library support, and general consensus on best practices. The industry support and lack of use cases for Macaroon's third-party caveats in common inter-service scenarios give JWTs and edge. The use of tokens to limit the effects of node or service compromise and to implement detailed resource level access policies satisfies the

functional requirement of granular authorization control. When the token minting is done centrally the use of asymmetric options reduces the need for secret management. As long as the choice of algorithm is RSA with longer keys than 3076 bits or ECDSA with key length longer than 256 bits the solution is secure (Smart 2018). In this architecture model, the authorization service becomes a single point of failure, the compromisation of which would compromise the whole system. On the other hand, when this is identified, extra hardening methods can be applied to the service, such as implementing additional security functionalities in the used container images and execution environment.

For the quality attributes the model satisfies horizontal load scalability as each service can be independently scaled with replication. Performance overhead is hard to gauge but the use of proxies to route traffic adds a networking hop compared to direct service-to-service communication. Though, the native implementation without a supplemental sidecar eliminates the need to communicate through localhost and can be optimized for maximum performance for that particular case (Burns 2018 p. 14).

Platform agnosticism is achieved by using orchestration that can be deployed on a variety of platforms and opting to not use platform specific tooling or writing an abstraction layers on top. The minimal code implementation criterion is a tough one. When opting for the native implementation without sidecar it means the code needs to be implemented by the service itself, which is bound to mean more application specific code. The duplication of engineering efforts can be reduced by developing shared libraries or agreeing on the use of certain existing ones.

7.2 Service Mesh with Management Layers

In this example model, container orchestration takes care of secure service instance hosting. MTLS connections are implemented through sidecar proxies such as NGINX or Envoy (NGINX 2018; Envoy 2018) based on certificates from an internal PKI injected to the sidecars from a central certificate store. The authorization logic is also implemented by sidecars based authorization policies fetched from a central policy source. The great thing from a developer's convenience point of view is that the token can be added to the requests without the application having to worry about it. The developer does not need to worry about access control as the mTLS implementation and the addition of bearer credentials is all done on the sidecar proxy. Though, configuring the sidecar to application needs still needs to be done initially.

Figure 8 below represents a generic service mesh model with a control plane that encapsulates the central policy source and certificate storage for injecting certificates from the internal PKI to the sidecars. Every connection between the services is secured with mTLS using the certficates.

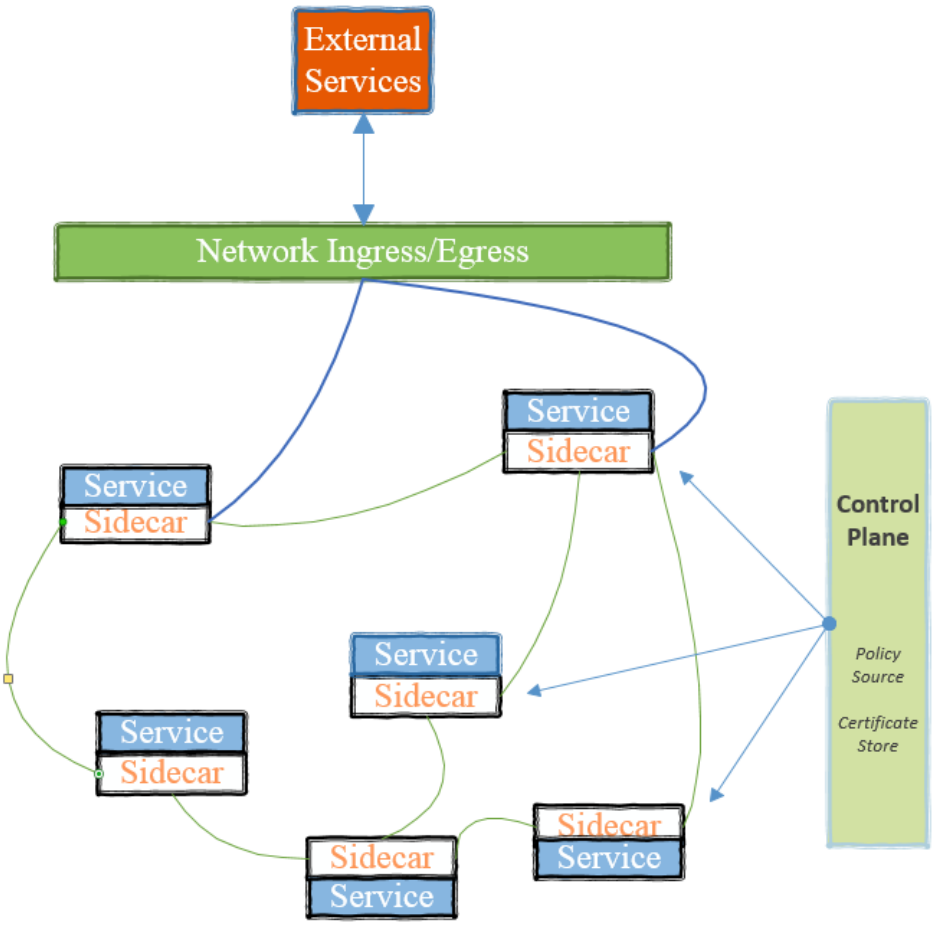

Figure 8 The Service Mesh Model with Management Layers

Service mesh shines in being able to provide features with modular sidecars without the application developers needing particular expertise in the area. From our perspective, the most important factor here would be the chance to implement mutual TLS connections between sidecars. Due to TLS termination happening on the proxy, and just the plain, unencrypted request, being passed to the actual service, the developer does not need to change the design or code of their application at all in order to gain encrypted communications between services and a lot of reusability that can speed up the development process (Burns 2018 p.16). The developer also does not need to be concerned about interoperability and secure algorithmic choices and implementation as much in regard to TLS and effort can be focused on differentiating service features. Based on empirical experience, major source code modifications always require a justification and a value proposition, which makes security features being offered for free hard to turn down.

For authorization in the service mesh both tokens and RBAC/ABAC solutions can be leveraged. If opting to use tokens and handling authorization logic on the sidecar proxy, it makes the most sense to use HMAC based tokens as both the signing and verification are done at the same place and do not in general need to be done anywhere else. This

allows for a performant solution without having to think about distribution and management of keys. The benchmark 128 bits of security can be achieved by using a SHA256 or stronger based HMAC. Both JWTs and macaroons can be used and the choice is based on whether it is considered more valuable to have a battle-tested solution that is easily used out of the box with JWTs or if the application can leverage the customization possibilities, delegation, and attenuation of macaroons. For authorization schemes based solely on embedded claims in certificates, the important part is to ensure the certificate is signed with secure algorithms and encapsulates strong enough identifiers to prevent identity spoofing to achieve at least 128 bits of security.

The service mesh and sidecar proxy model offer a lot of possibilities for optimizing performance of the networking side of things. As expertise in the software defined networking area is not necessarily something that the service developers possess, separating the development of functionalities allows different people to focus on their area of expertise. For the sidecar developers, it means freedom in their technology choices to optimize for their use case independently of the choices made with the application container (Richardson & Smith 2016, p. 8).

As a downside adding more components complicates a system. Especially the increased complexity is seen on the deployment side of things. This is where the orchestration systems and automated code defined deployments become vital to manage a complex system. In the **table 11** below, we have the distilled pros and cons of the service mesh approach.

Table 11. *Pros and Cons of the Service Mesh*

Pros	Cons
Allows for fine-grained resource management of the sidecars	Introduces some latency to calls in inter-process communication
Sidecars can be worked on and maintained independently	Complicates deployments
Optimization for a specific function with technologies of choice	One more component to maintain
Same sidecar can be used by services written in a variety of languages	Failure of sidecar renders dependent service behind it useless

7.3 Comparison of Models

Earlier we established a set of functional requirements and quality attributes that the system design candidates need to adhere to. When we consider the security properties of these systems against the situation in the beginning where all the communication between microservices in an application happen unencrypted in a private network (walled garden), we see a clear improvement. The adversary we defined and want to counter had breached the walled garden and was among other things able to eavesdrop on all the communication and tamper with the messages with no defense mechanisms working against them. To counter this, we established methods to protect communications between services with mutually authenticated transport layer security.

With mTLS and good security choices in used cryptographic algorithms, we achieve battle-tested ways to guarantee origin authenticity, message integrity and authenticity, and confidentiality in the communications, achieving at least 128 bits of security in each of these areas. For both systems, a major factor is the trust on the internal PKI. The integrity and security of the system hinges on the non-compromisation of this trusted party.

The second factor was what happens when a service gets compromised. In the initial scenario, there was little protection against extracting data from the services in the same network as implicit trust was put on all the entities in the VPC. With authorization controls, whether token or RBAC based, we limit the access to resources of each service to the minimum required to fulfill their role. Both proposed models have ways to achieve minimum 128 bits of security to guard from further unauthorized access to resources beyond the one compromise, whether embedded in the certificates leveraging internal CA or through the use of cryptographically signed bearer tokens carrying claims to ensure they are authentic and have not been tampered with.

The biggest difference between the models being that in a service mesh all the access control and networking is offloaded onto the sidecar proxies and the proxy mesh solution has to implement mTLS connections for service-to-proxy connections in application code. **Table 12** below presents how the two models adhere to the set functional requirements.

Table 12. *Adherence to the Functional Requirements*

Criteria	*Proxy Mesh*	*Service Mesh*
Service-to-service communications are encrypted	MTLS is provided through native implementation and inter-service proxy	MTLS established on sidecar proxy-to-sidecar proxy connections.

Service-to-service calls are authenticated	Certificates from internal CA. Application code implementation of service-to-proxy and proxy-to-service mTLS.	Certificates through an internal CA. MTLS on all connections.
Granular access control	Central authorization service enforces per-service policies that can be very granular. Implementation through JWT bearer tokens.	RBAC, ABAC, and token implementations are possible. Each service sidecar enforces their own policy.
Deployment of the system can be defined in code	Through orchestration solution	Through orchestration solution

On the other hand, when considering the set quality requirements for the system, we start seeing more difference between the proposed system architectures. The quality attributes are usually the harder ones to gauge as well but some general analysis can be done. The first quality attribute we set was minimal performance overhead compared to a system without these security properties. The performance implications are hard to approximate without practical system benchmarks. Bagge & Yarygina (2018) proposed that adding mTLS and user principal propagation with tokens represented a total of 11% overhead. On the other hand, Istio performance documentation (2018) asserts that mutual TLS costs are negligible on hardware supporting the Advanced Encryption Standard New Instructions (AES-NI). The instruction set enables the use of hardware to accelerate parts of the cryptographic operations used in TLS. Though the utilization of this technology requires access to the real cores and thus is not applicable to virtualized environments in public clouds (Calomel 2018).

More detailed performance implications are dependent on the practical implementation. If for example OpenSSL is used for TLS, the core libraries are implemented in C (OpenSSL 2018) and the calling programming language has lesser impact on performance. Sidecar proxy can focus on a higher performing language for micro-optimization of networking operations but similarly the proxies in a proxy mesh can. Developers can fine-tune the performance and security scale with their choice of cryptographic algorithms for example.

Horizontal load scalability works the same way in both models, by replicating service instances (Dragoni et al. 2018). In the service mesh model this means instantiation of an application container and a sidecar container along it, in the proxy mesh solution the scaling can happen on either the proxy instances or service instances depending on the needs.

Having more options for fine-tuning the deployment scalability gives a slight edge to the proxy mesh solution, though fine-tuning and optimization comes with the requirement for greater operational finesse.

The quality attribute of minimal code changes required favors the service mesh solution as it offloads many functionalities to a sidecar which promotes reusability with several services (Burns 2018 p.16 - 17). In the proxy mesh solution where application containers are stand-alone, code changes are required to introduce mTLS and access controls for a service. The sidecar being generic enough and usable by myriad of services gives the clear edge to service mesh solution. Shared libraries can offer modularity and re-usability for native implementations but this requires at least wrappers for libraries to support different programming languages to leverage the benefit of allowing different services to be developed in different languages to increase developer efficiency. As the available computing resources grow and become cheaper, the developer efficiency and convenience factors become more important, which favor the service mesh model.

Cloud agnosticism factor is the same for both systems as neither of them relies on platform specific tools. The degree of agnosticism is dependent on the container orchestration and deployment tooling. This means both models in principal allow for the benefits of multi-cloud deployments, such as computing price arbitrage, increased resiliency, and low latency coverage for a global userbase (Guerrero et al. 2018). Low platform coupling also means that the services can in the optimal case utilize a hybrid-cloud paradigm with dedicated bare metal resources, as well as public clouds using the same methods, the range of options being limited by the choice of deployment system and available tooling. The **table 13** lists the distilled benefits and drawbacks of the models.

Table 13. *Pros and Cons of the Proxy and Service Mesh Models*

Service Mesh	**Proxy Mesh**
General use sidecars for different needs	Greater optimization with native code
Existing well developed sidecar proxies	Larger customization required and allowed
Out of the box comprehensive solutions available, modifying still requires a lot of effort	Engineering effort required to piece together a comprehensive system, out of the box solutions not readily available

8. CONCLUSIONS

Microservices with containers promise increased operational efficiency, shorter software development lifecycles, and independent service scaling that lends itself well to automation and code-defined infrastructure (Trihinas et al. 2018; Dragoni et al. 2017). We identified, based on existing literature and our own analysis, several areas of security concerns that have arisen along the new paradigm. These concerns for the most part had not been assessed thoroughly in existing grey or academic literature.

Microservice security problems can be solved in various ways but all require identifying and deliberate effort to answer them. Microservice communications defense-in-depth methods are a combination between technical measures on various levels of depth, and good architectural choices to enable the benefits of the paradigm without hampering security. The traditional idea of network security controls is not sufficient anymore to protect microservice applications, which are often deployed in a hybrid-cloud environment, mixing different public clouds and dedicated datacenter resources. That is why we started with the assumption of the wall being broken and the adversary being inside the network.

We started by identifying the three layers of microservice security stack. These were container orchestration, communications layer, and the service layer. We discussed how container orchestration can be leveraged to provide secure deployment of containers hosting our services. Kubernetes and Docker SwarmKit were analyzed for their security measures and we discovered that SwarmKit offers security-by-default while the more extensive platform Kubernetes offers a mixed bag of insecure and secure options, putting more responsibility on the developer to make secure choices.

To offer security in the communications layer we discussed mutual TLS as a way of achieving a secure channel that provides strong guarantees of origin and message authenticity, confidentiality, replay protection, and integrity. With container orchestration, we can establish mTLS connections in a container cluster. Service level mTLS can be implemented with a node pattern such as a reusable colocated sidecar container or implement it in the service source code. From individual service level, we moved up in abstraction to assess system level patterns and identified strengths and weaknesses of proxy gateway, proxy mesh, and service mesh models in security context. We eliminated the proxy gateway from further consideration due to its coupling and performance implications.

After establishing ways to achieve a secure communication channel between services, we focused on deeper security considerations to reduce the effect of service compromise to its minimum. This was done by evaluating different authorization schemes from access control lists to bearer tokens encapsulating claims on resources. We presented two models for organizing authorization with a central authorization service and one option for completely per-service independent authorization with sidecars.

Lastly, we pieced together all the assessed levels of security into a comprehensive proxy mesh and service mesh models that offer defense-in-depth security considerations and counter the adversary we defined in the beginning. The two models were analyzed based on a set of functional and quality requirements as per the ATAM evaluation model to recognize their strengths and environments they best fit in. For the most part, their security properties were the same and the differences were in engineering effort and reusability of the chosen methods. **Table 14** below summarizes the security features of the two models and what is the minimum required strength for each feature to provide an acceptable level of security of over 128 bits of security.

Table 14. *Summary of the Proxy Mesh and Service Mesh Models*

Level	**Proxy Mesh**	**Service Mesh**	**Security**
Authorization	JWTs, Macaroons	JWTs, ABAC, RBAC	HS256 or >= 3076 bit RSA or >= 256 bit ECDSA
Authentication	Native mTLS implementation in services	Sidecar implements mTLS along other networking functionality	Identity based on certificates with >= 256 bit ECDSA or >= 3076 bit RSA TLS key lengths >= 128 bits of security
Container orchestration	Kubernetes/Docker SwarmKit		Node introduction with tokens of >= 128 bits of security Node identity >= 128 bits of randomness

The thesis contributed new analysis on defense-in-depth methods in service-to-service communications in microservice applications and maps out the layers where security concerns need to be taken into account in deploying these applications. We also pieced together areas of microservice application security concerns that have been discussed separately in academia and industry to a holistic view of microservice communications security for REST applications deployed on hybrid cloud environments. As a result, we have described a system that employs deeper security controls on several levels and counters an adversary that has breached a virtual private cloud. The thesis ultimately serves as a resource for designing practical and secure microservices application architectures.

The critical points for the reproducibility of the results of this thesis are the choice of literature and models. What should be considered the established best practices boil down to a subjective view of the writers as objective criteria for evaluation is very hard to come

by due to different organizational and personal preferences. Disagreement with the results are bound to come by and are welcomed in order to further the field.

8.1 Further Research

As the thesis looks at the topic from a quite high-level view, future research extending the ideas presented here can focus on any of the sub-concerns. For example, the security concerns of container orchestration or the various service mesh implementations have not been analyzed in depth in academic literature. The solutions available in the industry are progressing at a fast pace and enterprises are looking to adapt the new developments to reap the rewards. To support organizations making good decisions security-wise, more knowledge and further analysis is required.

An interesting area of research would also be to look more deeply into the principal propagation between services, an area that was touched on briefly in this thesis. Also, the policy definition language and best practice exploration for the propagation with macaroons and JWTs would be very interesting to look into. In particular, as the standardization for macaroons, despite their potential, is lacking, a cross-disciplined look combining security and linguistics research into defining caveat language further would make them more easily approachable and utilizable.

From a practical point of view, an industry-aligned exploration of the available multi-cloud deployment tooling and their security concerns would be of much use to several enterprises tackling the issues rising from a move into microservices world. This research could be complemented with analysis of the gained benefits of multi-cloud use when it comes to price of computing, performance, and service robustness and availability.

REFERENCES

Abadi M. & Needham R. (1994). Prudent Engineering Practice for Cryptographic Protocols. Research in Security and Privacy, 1994. Proceedings., 1994 IEEE Computer Society Symposium on Research in Security and Privacy. Available at: 10.1109/RISP.1994.296587

Aggarwal D., Maurer U. (2008). Breaking RSA Generically is Equivalent to Factoring. Available at: https://eprint.iacr.org/2008/260.pdf

Akamai. (2017). Akamai Online Retail Performance Report: Milliseconds Are Critical" Available at: https://www.akamai.com/us/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp

Alshuqayran N., Ali N., Evans R. (2016). A Systematic Mapping Study in Microservice Architecture. 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA). Available at: http://eprints.brighton.ac.uk/16235/1/PID4474889.pdf

Amazon. (2018). Netflix Case Study. Available at: https://aws.amazon.com/solutions/case-studies/netflix/

Amazon Simple Storage Service. (2018). Access Control List (ACL) Overview. AWS Documentation. Available at: https://docs.aws.amazon.com/AmazonS3/latest/dev/acl-overview.html

Anderson R.J. (2008). Security Engineering: A Guide to Building Dependable Distributed Systems. Chapter 4.2.2. Second Edition. Wiley Publishing, Inc.

Barker E. Recommendation for Key Management Part 1: General. NIST Special Publication 800-57 Part 1 Revision 4. Available at: http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4

Bass, L., M.H., Klein, G., Moreno, (2001). Applicability of General Scenarios to the Architecture Tradeoff Analysis Method. Software Engineering Institute, Carnegie Mel- lon University, Pittsburgh, Pennsylvania Tech. Rep. CMU/SEI-2001-TR-014. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5637

Barenghi A., Mainardi N., Pelosi G. (2018). Systematic Parsing of X.509: Eradicating Security Issues with a Parse Tree. Journal of Computer Security, Volume 26, Issue 6, 30th October 2018. Available at: https://arxiv.org/abs/1812.04959

Bellare M., Canetti R. & Krawczyk H. (1996). Keying Hash Functions for Message Authentication. Available at: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=BD49250080E6F70F1DD7C9EBF626635E?doi=10.1.1.134.8430&rep=rep1&type=pdf

Bellare M. & Tackmann B. (2016). The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3. CRYPTO 2016: Advances in Cryptology – CRYPTO 2016 pp. 247-276. Lecture Notes in Computer Science, vol 9814. Springer, Berlin, Heidelberg. Available at: https://doi.org/10.1007/978-3-662-53018-4_10

Bernstein J., Duif N., Lange T., Schwabe P. Yang B. High-speed high-security signatures. Journal of Cryptographic Engineering 2 (2012), 77-89. Available at: https://cr.yp.to/papers.html#ed25519

Birgisson A., Politz Gibbs J., Erlingsson U., Taly A., Vrable M., Lentczner M. (2014). Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. Available at: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/41892.pdf

Boneh D. & Shoup V. (2017). A Graduate Course in Applied Cryptography. P. 595 – 597. Version 0.4. Available at: https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_4.pdf

Braun E., Schlachter T., Düpmeier C., Stucky KU., Suess W. (2017) A Generic Microservice Architecture for Environmental Data Management. In: Hřebíček J., Denzer R., Schimak G., Pitner T. (eds) Environmental Software Systems. Computer Science for Environmental Protection. ISESS 2017. IFIP Advances in Information and Communication Technology, vol 507. Springer, Cham. Available at: https://doi.org/10.1007/978-3-319-89935-0_32

Burns B. (2018). Designing Distributed Systems. O'Reilly Media. First Edition.

Calomel. (2018). AES-NI SSL Performance, a study of AES-NI acceleration using LibreSSL, OpenSSL. Accessed at: 20.12.2018. Available at: https://calomel.org/aesni_ssl_performance.html

Cooper D., Santesson S., Farrell S., Boeyen S., Housley R., Polk W. (2008). "Internet x.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile". Internet Engineering Task Force. Request for Comments: 5280. Available at: https://tools.ietf.org/html/rfc5280#page-9

Consul. (2018). ACL System. Consul Documentation. Available at: https://www.consul.io/docs/guides/acl.html

Costa B., Pires P.F., Delicatio F.C., Merson P. (2014). Evaluating a Representational State Transfer (REST) Architecture: What is the Impact of REST in My Architecture?. *IEEE/IFIP Conference on Software Architecture*, Sydney, NSW, 2014, pp. 105-114. Available at: https://ieeexplore.ieee.org/document/6827107

Costa B., Pires P.F., Delicatio F.C., Merson P. (2016). Evaluating REST Architectures – Approach, tooling, and guidelines. Journal of Systems and Software Volume 112. Pages 156-180. Available at: https://www.sciencedirect.com/science/article/pii/S0164121215002150

Costan V. & Devadas S. (2016). Intel SGX Explained. Cryptology ePrint Archive: Report 2016/086. Available at: https://eprint.iacr.org/2016/086

Cremers C., Horvat M., Hoyland J., Scott S., van der Merwe T. (2017). A Comprehensive Symbolic Analysis of TLS 1.3. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security Pages 1773-1788. Available at: https://dl.acm.org/citation.cfm?id=3134063

Davies J., Schorow D., Samrat R., Rieber D. (2008). SOA Security. In: The Definitive Guide to SOA. Pp. 225 – 264. Apress.

Digicert. (2018). SSL Certificate Validation Process from DigiCert. Available at: https://www.digicert.com/ssl-validation-process.htm

Docker SwarmKit (2018). Swarmkit Source Code. Accessed: 30.11.2018. Available at: https://github.com/docker/swarmkit

Docker Swarm (2018). Swarm mode overview. Accessed: 27.11.2018. Available at: https://docs.docker.com/engine/swarm/

Dodis Y. et al. (2013). Security analysis of pseudo-random number generators with input: /dev/random is not robust. CCS '13 Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. Available at: https://dl.acm.org/citation.cfm?id=2516653

Dragoni N. Giallorenzo S., Liuch L. A., Mazzara M., Montesi F., Mustafin R., Safina L. Microservices: Yesterday, Today, and Tomorrow. (2017). Present and Ulterior Software Engineering pp 195 – 216, Springer, Cham. Available at: https://doi.org/10.1007/978-3-319-67425-4_12

Dragoni N., Lanese I., Larsen S.T., Mazzara M., Mustafin R., Safina L. (2018) Microservices: How to Make Your Application Scale. In: Petrenko A., Voronkov A. (eds) Perspectives of System Informatics. PSI 2017. Lecture Notes in Computer Science, vol 10742. Springer, Cham. Available at: https://doi.org/10.1007/978-3-319-74313-4_8

Envoy. What is Envoy. (2018). Available at: https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy

Ferraiolo D.F. & Kuhn D.R. (1992). Role-Based Access Controls. 15th National Computer Security Conference (1992). Pp. 554 – 563.

Fielding R.F., Architectural Styles and the Design of Network-based Software Architectures (Ph.D. dissertation). University of California, Irvine, (2000). Available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Fielding R., Reschke J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Authentication. Internet Engineering Task Force, Request for Comments: 7235. Available at: https://tools.ietf.org/html/rfc7235

Fisher B. (2018). It's 2018, Is Swarm Dead? Answered by a Docker Captain. Available at: https://www.bretfisher.com/is-swarm-dead-answered-by-a-docker-captain/

Fu P., Li Z., Xiong G., Cao Z., Kang C. (2018). SSL/TLS Security Exploration Through X.509 Certificate's Life Cycle Measurement. 2018 IEEE Symposium on Computers and Communications (ISCC). Available at: 10.1109/ISCC.2018.8538533

Gajek S., Manulis M., Pereira O., Sadeghi A., Schwenk J. (2008). Universally Composable Security Analysis of TLS. 2nd International Conference on Provable Security, ProvSec 2008, LNCS 5324. Available at: https://eprint.iacr.org/2008/251

Gellman B., Soltani A. (2013). NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say. The Washington Post, October 30, 2013. Available at: https://www.washingtonpost.com/world/national-security/nsa-infiltrates-links-to-yahoo-google-data-centers-worldwide-snowden-documents-say/2013/10/30/e51d661e-4166-11e3-8b74-d89d714ca4dd_story.html

Google Cloud. (2016). Spotify Chooses Google Cloud to Power Data Infrastructure. Google Cloud Platform Blog. Available at: https://cloudplatform.googleblog.com/2016/02/Spotify-chooses-Google-Cloud-Platform-to-power-data-infrastructure.html

Google. (2017). Google Infrastructure Security Design Overview. Available at: https://cloud.google.com/security/infrastructure/design/

Guerrero, C., Lera, I. & Juiz, C. J. (2018). Resource Optimization of Container Orchestration: A Case Study in Multi-Cloud Microservices-based Applications. The Journal of Supercomputing (2018) 74: 2956. Available at: https://doi.org/10.1007/s11227-018-2345-2

Hardt D. (2012). The OAuth 2.0 Authorization Framework. Internet Engineering Task Force Request for Comments 6749. Available at: https://tools.ietf.org/html/rfc6749

Hardy N. 1988. The Confused Deputy (or why capabilities might have been invented). Operating Systems Reviews, Vol 22. Available at: http://cap-lore.com/CapTheory/ConfusedDeputy.html

Hu V.C., Ferraiolo D., Kuhn R., Schnitzer A., Sandlin K., Miller R., Scarfone K. (2014). Guide to Attribute Based Access Control (ABAC) Definition and Considerations. NIST Special Publication 800-162. National Institute of Standards and Technology, U.S Department of Commerce. Available at: http://dx.doi.org/10.6028/NIST.SP.800-162

Hühn T. (2018). Myths about /dev/urandom. Available at: https://www.2uo.de/myths-about-urandom#orthodoxy

Hykes. S. (2017). Docker Platform and Moby Project Add Kubernetes. Docker Blog. Available at: https://blog.docker.com/2017/10/kubernetes-docker-platform-and-moby-project/

Izrailevsky Y. & Tseitlin A. (2011). The Netflix Simian Army. The Netflix Tech Blog. Available at: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116

Istio. (2018). Performance and Scalability. Istio Concepts. Accessed at: 8.12.2018. Available at: https://istio.io/docs/concepts/performance-and-scalability/

Istio. (2018). Security. Istio Documentation. Available at: https://istio.io/docs/concepts/security/#authorization

Jander K., Braubach L., Pokahr A. (2018). Defense in-depth and Role Authentication for Microservice Systems. The 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018). Available at: https://www.sciencedirect.com/science/article/pii/S1877050918304009

Jhawar R., Piuri V., Santambrogio M. (2013). Fault Tolerance Management in Cloud Computing: A System-Level Perspective. IEEE Systems Journal p. 288-297, Volume 7, issue 2, June 2013. Available at: 10.1109/JSYST.2012.2221934

Jones M., Bradley J., Sakimura N. (2015). JSON Web Token (JWT). Internet Engineering Task Force RFC 7519. Available at: https://tools.ietf.org/html/rfc7519

JWT. (2018). Libraries for Token Signing/Verification. Accessed at: 12.10.2018. Available at: https://JWT.io

Kazman R., Klein M., Clements P. (2000). ATAM: Method for Architecture Evaluation. Carnegie Mellon Software Engineering Institute. Available at: https://resources.sei.cmu.edu/asset_files/TechnicalReport/2000_005_001_13706.pdf

Kang H., Le M., Tao S. (2018). Container and Microservice Driven Design for Cloud Infrastructure DevOps. 2016 IEEE International Conference on Cloud Engineering (IC2E). Available at: https://ieeexplore.ieee.org/document/7484185/

Klein M. (2017). Service mesh data plane vs. control plane" Available at: https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc

Krawczyk H., Bellare M. & Canetti R. Internet Engineering Task Force. (1997). RFC 2104 HMAC: Keyed-Hashing for Message Authentication. Available at: https://www.ietf.org/rfc/rfc2104.txt

Kubernetes. (2018a). Website. Accessed at: 30.11.2018. Available at: https://kubernetes.io/

Kubernetes. (2018b). Authenticating. Kubernetes Documentation. Available at: https://kubernetes.io/docs/reference/access-authn-authz/authentication/#static-token-file

Kubernetes. (2018c). Pod Overview. Kubernetes Documentation. Available at: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

Kubernetes. (2018d). TLS bootstrapping. Kubernetes Documentation. Available at: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-tls-bootstrapping/

Kubernetes Community. (2018a). Super Simple Discovery API. 2018. Accessed at: 28.11.2018. Available at: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/cluster-lifecycle/bootstrap-discovery.md

Kubernetes Community. (2018b). Source code. Accessed at: 28.11.2018. Available at: https://github.com/kubernetes/kubernetes

Lauter K. E. & Stange E. K. (2008). The elliptic curve discrete logarithm problem and equivalent hard problems for elliptic divisibility sequences. Lecture Notes in Computer Science Volume 5381 (2009) 309-327. Available at: https://arxiv.org/abs/0803.0728

Lewis J. (2012). Micro Services – Java, the Unix Way. 33rd Degree Conference. Available at: http://2012.33degree.org/talk/show/67

Linden, M. (2017). Identiteetin- ja pääsynhallinta. [Identity and Access Management]. Tampereen teknillinen yliopisto. Tietotekniikan laboratorio. Raportti; Vuosikerta 7). Tampere University of Technology.

Linkerd (2018). Linkerd Proxy. Accessed at: 1.12.2018. Available at: https://linkerd.io/1/features/http-proxy/

Linux. (2017). GETRANDOM(2). Linux Programmer's Manual. Available at: http://man7.org/linux/man-pages/man2/getrandom.2.html

Louis R. (2015). gRPC Motivation and Design Principles. Available at: https://grpc.io/blog/principles

Madan B.B., Goševa-Popstojanova, Vaidyanathan K., Trivedi K.S. (2004). A method for modeling and quantifying the security attributes of intrusion tolerant systems. Performance Evaluation, Volume 56, Issues 1-4, March 2004, pp.167-186. Available at: https://doi.org/10.1016/j.peva.2003.07.008

Mao W. (2002). A Structured Operational Modelling of the Dolev-Yao Threat Model. 10th International Workshop on Security Protocols. Pp. 34-36. Available at: https://doi-org.libproxy.tut.fi/10.1007/978-3-540-39871-4_5

McCaffrey J. (2016). Running Online Services At Riot: Part I. Available at: https://engineering.riotgames.com/news/running-online-services-riot-part-i

McGrew D. (2008). An Interface an Algorithms for Authenticated Encryption. IETF Network Working Group, Request for Comments: 5116. Available at: https://tools.ietf.org/html/rfc5116

McLean T. (2015). Critical Vulnerabilities in JSON Web Token Libraries. Available at: https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

Menezes A.J., van Oorschot P.C. & Vanstone S.A. (1997). Handbook of Applied Cryptography. CRC Press. Pages 283, 286

Microsoft. (2017a). Containerized Microservices. Available at: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/containerized-microservices

Microsoft. (2017b). Sidecar pattern. Available at: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

Microsoft. (2003). Understanding SOAP. Available at: https://msdn.microsoft.com/en-us/library/ms995800.aspx

Monica D. (2017). "Least Privilege Container Orchestration" https://blog.docker.com/2017/10/least-privilege-container-orchestration/

National Institute of Standards and Technology. (2015). Secure Hash Standard. FIPS PUB 180-4. Available at: http://dx.doi.org/10.6028/NIST.FIPS.180-4

National Institute of Standards and Technology. (2013). Digital Signature Standard (DSS). FIPS PUB 186-4. Available at: http://dx.doi.org/10.6028/NIST.FIPS.186-4

National Institute of Standards and Technology. (2012). Recommendation for Applications Using Approved Hash Algorithms. NIST Special Publication 800-107. Available at: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf

National Institute of Standards and Technology (2016). CVE-2016-10555 Detail. Available at: https://nvd.nist.gov/vuln/detail/CVE-2016-10555#vulnCurrentDescriptionTitle

National Vulnerability Database. (2014). CVE-2014-0160 Detail. Available at: https://nvd.nist.gov/vuln/detail/CVE-2014-0160

National Vulnerability Database. (2016). CVE-2016-10555 Detail. Available at: https://nvd.nist.gov/vuln/detail/CVE-2016-10555

NGINX. (2018). NGINX Reverse Proxy https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/

Nir Y. & Langley A. (2018). ChaCha20 and Poly1305 for IETF Protocols. Internet Research Task Force (IRTF). Request for Comments: 8439. Available at: https://tools.ietf.org/html/rfc8439

NPM (2018). jwt-simple. Available at: https://www.npmjs.com/package/jwt-simple

OpenSSL. (2018). Source Code. Github. Accessed at: 30.11.2018. Available at: https://github.com/openssl/openssl

Osnat R. (2018). Which Kubernetes Platform is Right for Your Enterprise? Available at: https://blog.aquasec.com/kubernetes-management-platform-for-the-enterprise

Otterstad C., T. Yarygina. (2017). Low-Level Exploitation Mitigation by Diverse Microservices. ESOCC 2017: Service-Oriented and Cloud Computing. Lecture Notes in Computer Science, vol 10465. pp. 49–56. Springer, Cham. Available at: https://dx.doi.org/10.1007/978-3-319-67262-5_4

Phan C. (2007). Service Oriented Architecture (SOA) – Security Challenges and Mitigation Strategies. Milcom 2007 – IEEE Military communications Conference. Available at: 10.1109/MILCOM.2007.4455012

ProgrammableWeb. (2019). Search the Largest API Directory on the Web. Available at: https://www.programmableweb.com/apis/directory

Rescorla E. (2000). HTTP Over TLS. Internet Engineering Task Force. Request for Comments: 2818. Available at: https://tools.ietf.org/html/rfc2818

Rescorla E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. Internet Engineering Task Force. Request for Comments: 8446. Available at: https://tools.ietf.org/html/rfc8446

Richardson C., Smith F. (2016). Microservices from Design to Deployment. NGINX.

Richardson L., Ruby S. (2007): RESTful Web Services. O'Reilly Media, First Edition.

Rivest R.L., Shamir A., Adleman L. (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM. Volume 21 Issue 2, Feb. 1978, p. 120-126. Available at: https://dl.acm.org/citation.cfm?id=359342

Saltzer, J.H. & Schroeder, M.D. (1975). The Protection of information in computer systems. *Proceedings of the IEEE,* vol. 63, no. 9 (Sept 1975), pp. 1278-1308. Availabe at: http://web.mit.edu/Saltzer/www/publications/protection/

Schneider F. B. (2003). Least Privilege and More. Cornell University. Available at: https://www.cs.cornell.edu/fbs/publications/leastPrivNeedham.pdf

Schneier B. (1996). Applied Cryptography: Protocols, Algorithms and Source Code in C. John Wiley & Sons, Inc. Chapter 18.14

Scoldani J., Tamburri D.A., Van Den Heuvel W. (2018). The pains and gains of microservices: A Systematic grey literature review. Journal of Systems and Software Volume 146, Pages 215-232. Available at: https://www.sciencedirect.com/science/article/pii/S0164121218302139

Shannon C. (1949). Communication Theory of Secrecy Systems. Bell System Technical Journal. 28. 662. Available at: 10.1002/j.1538-7305.1949.tb00928.x

Smart N.P. (2013). Cryptography, An Introduction: Third Edition. Third Edition. Available at: http://people.cs.bris.ac.uk/~nigel/Crypto_Book/

Smart N.P. et al. (2018). Algorithms, Key Size and Protocols Report (2018). ECRYPT – Coordination & Support Action. Available at: http://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf

Sousa G. Rudametkin W., Duchien L. (2016). Automated Setup of Multi-Cloud Environments for Microservices Applications. 2016 IEEE 9th International Conference on Cloud Computing. Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7820288&tag=1

Souppaya M., Morello J., Scarfone K. (2017). Application Container Security Guide. NIST Special Publication 800-190. Available at: https://doi.org/10.6028/NIST.SP.800-190

Spomky Labs. (2018). Benchmarks, Result Table. Accessed at: 5.12.2018. Available at: https://web-token.spomky-labs.com/benchmarks/result-table

Stetson C. (2018). Microservices Reference Architecture. NGINX.

Tankeresley G. (2018). Kubelet TLS bootstrap. Kubernetes Project, Design Proposal. Accesse at: 30.11.2018 Available at: https://github.com/kubernetes/community/blob/b3349d5b1354df814b67bbdee6890477f3c250cb/contributors/design-proposals/cluster-lifecycle/kubelet-tls-bootstrap.md

Topalovic E., Saeta B., Huang L.S., Jackson C. & Boneh D. (2012). Towards Short-Lived Certificates. Proceedings of IEEE Oakland Web 2.0 Security and Privacy. Available at: http://www.ieee-security.org/TC/W2SP/2012/papers/w2sp12-final9.pdf

Trihinas D., Tryfonos A., Dikaiakos M. D., Pallis G. (2018). DevOps as a Service: Pushing the Boundaries of Microservice Adoption. in *IEEE Internet Computing*, vol. 22, no. 3, pp. 65-71, May./Jun. 2018. Available at: https://doi.org/10.1109/MIC.2018.032501519

Van Bulck et al. (2018). FORESHADOW: Extracting the Keys to the Intel SGZ Kingdom with Transient Out-of-Order Execution. Proceedings of the 27th USENIX Security Symposium. Available at: https://foreshadowattack.eu/foreshadow.pdf

Walsh K. & Manferdelli J. (2017). Intra-Cloud and Inter-Cloud Authentication. 2017 IEEE 10th International Conference on Cloud Computing. Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8030604

Yarygina .T, RESTful is not secure. (2017). in Applications and Techniques in Information Security (ATIS 2017). Springer, 2017, pp. 141–153. Available at: https://link.springer.com/book/10.1007%2F978-981-10-5421-1

Yarygina T. & Bagge A.H. (2018). Overcoming Security Challenges in Microservice Architectures. 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE). Available at: https://ieeexplore.ieee.org/document/8359144