Aalto University
School of Science
Bachelor's Programme in Science and Technology

Security in Microservice Architecture — Impact of the Switch from Monolith to Microservices

Bachelor's Thesis

18. joulukuu 2020

Tommi Jäske

Aalto-yliopisto
Perustieteiden korkeakoulu
Teknistieteellinen kandidaattiohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

Tekijä:	Tommi Jäske
Työn nimi:	Turvallisuus mikropalveluarkkitehtuurissa - Monoliitisesta arkkitehtuurista siirtyminen mikropalveluarkkitehtuuriin ja sen vaikutukset.
Päiväys:	18. joulukuu 2020
Sivumäärä:	27
Pääaine:	Computer Science
Koodi:	SCI3027
Vastuuopettaja:	Professori Eero Hyvönen
Työn ohjaaja(t):	Professori Tuomas Aura (Tietotekniikan laitos)

Verkkopalvelut on usein toteutettu käyttäen monoliittista arkkitehtuuria. Tämä arkkitehtuuri ei skaalaudu eikä mahdollista ketterää kehitystä. Mikropalveluarkkitehtuurin käyttö mahdollistaa nämä sekä useita muita etuja.

Työn aiheena on tietoturva siirryttäessä monoliittisesta arkkitehtuurista mikropalveluarkkitehtuuriin verkkopalveluissa. Työn tarkoituksena oli selvittää keskeisimmät tietoturvakysymykset arkkitehtuurin vaihdoksessa ja esittää löydettyihin ongelmakohtiin ratkaisuja.

Aineistona käytettiin artikkeleja sekä alan peruskirjallisuutta. Työ toteutettiin kirjallisuustutkimuksena.

Tietoturvahaasteita ovat palvelun sisäinen viestintä, saavutettavuus ja skaalautuvuus, ajoympäristön tietoturva ja käyttäjän tunnistaminen ja valtuuttaminen.

Työssä havaittiin, että mikropalveluarkkitehtuuriin siirtymisessä on merkittäviä tietoturvariskejä, jotka tulee huolellisesti eritellä ja ratkaista jokaisessa arkkitehtuurin vaihdoksessa tapauskohtaisesti. Tietoturvan suunnittelu ja toteutus tulee suorittaa suurta huolellisuutta noudattaen ja mahdollisuuksien mukaan valmiita toteutuksia käyttäen.

Avainsanat:	monoliitti, mikropalvelu, arkkitehtuuri, tietoturva, vaihto
Kieli:	Suomi

Aalto University
School of Science
Bachelor's Programme in Science and Technology

ABSTRACT OF
BACHELOR'S THESIS

Author:	Tommi Jäske
Title of thesis:	Security in Microservice Architecture - Impact of a Switch from Monolith to Microservices
Date:	December 18, 2020
Pages:	27
Major:	Computer Science
Code:	SCI3027
Supervisor:	Professor Eero Hyvönen
Instructor:	Professor Tuomas Aura (Department of Computer Science)

There are many web services in use which were designed and implemented before the onslaught of microservices, using monolithic architectural style. This style does not scale nor lend it self to agile development practices, as well as the microservices architectural style does. Thus, these services might benefit from architectural change to using microservices. Since the architectures are very different, the security aspects need to be considered.

The topic of this bachelor's thesis is the security in the microservice architecture when switching from a monolithic architecture to microservices. The thesis presents the key security considerations and solutions to the major security issues.

The research is carried out as a literary study.

The main security considerations are communication, configuration, and authentication and authorization. Security must be considered as early as possible in a project and the use of existing tools and solutions should be encouraged.

Keywords:	monolith, microservice, architecture, security, switch
Language:	English

Contents

1 Introduction

In recent years, mobile applications and the web services which cater to them have revolutionized our daily lives by infiltrating social life, shopping and almost every aspect of our existence. The rapid expansion and, at times, even faster decline of these web services needs a matching architecture to meet their very specific needs.

One possible solution to address these architectural needs is the microservice architecture (MSA). There are multiple definitions for MSA but in general the web service is implemented using multiple small, independent services that act together. Newman (2019) definition for a microservice is a service that: is independently deployable, is modeled around business domain, that owns the data that they need to operate, that communicates via network, is technology agnostic, that encapsulates data storage and retrieval and that has a stable interface.

There are many web services already in use which were designed and implemented before the onslaught of microservices. Some of these services have already made the switch such as Netflix, but this is not the case for the whole industry.

When new development is carried out by a startup, the initial architecture might still be a monolith one. Also, when a new service is being created, the business domain might not be established yet. Additionally, there might exist a fair amount of uncertainty in what exactly is to be developed. Newman (2019) states that, due to limited software development resources, a monolith might be a better fit to the companies that are still trying to navigate their way to the product they eventually will offer. In the case of success, the need to rapidly scale up the offering emerges. Newman (2019) refers to these companies as "scale-ups". Newman (2019) also states that it is much easier to refactor an existing service than to create a new one, and thus the need to split monoliths to microservices is and probably will be relevant to the near future. This need is further amplified by the agile software development methods in which changes in requirements are welcome even in the later stages of the development process (Beck et al., 2001).

Kalske et al. (2018) find that, as the code base becomes large, the monolith architecture (MA) leads to slower development. This is due to the complexity inherent in the entwined monolith. As the development activity becomes more time consuming, more developers are needed to complete any changes to the code base.

Another reason for converting existing applications to microservices is the availability of skilled developers. New developers entering the workforce have a very different mindset than the older more seasoned professionals. Thus, it is clear that the ways of working and paradigms used are constantly changing. The Stack Overflow annual survey (Stack Overflow) conducted on developers found that half of the respondents identified as

full-stack or back-end developers. 40% of the respondents had less than five years of professional experience. Thus, the manpower that would be able and willing to keep the old monoliths running is not available.

Microservices are not the proper choice for all web services (Newman, 2019). However, microservices offer multiple benefits such as easier scalability and more modular structure for the application. When the architecture needs to be changed, the process needs to happen in an orderly and safe way. Also, often initially overlooked security aspects need to be identified and addressed as early as possible.

The microservice architecture (MSA) differs in many ways from the more traditional monolithic architecture (MA). When the architecture is changed, this shift entails very specific security issues. In this thesis, the MSA and related security literature is surveyed and the main differences between MA and MSA on security aspects are discussed.

Implementing security is hard. A theoretical proof for this can be found in Anderson (2001). His main finding was that an attacker has an advantage over the developers who are trying to defend a system. For the system to be secure, the defending developers must find all the bugs, whereas the attacker needs to find only one. Therefore, to make systems more secure, the use of existing tools and frameworks that have been tested and are already in use is preferable to developing one's own solution for a well-known and already solved problem.

The thesis is organized as follows. The next chapter compares the two architectures and the third chapter discusses the changing of architecture from MA to MSA. The fourth chapter discusses authentication and authorization. The fifth chapter discusses communication. In the seventh chapter, other relevant security concerns are discussed. The eigth and lastchapter in this thesis presents the conclusions and outlines further research topics.

2 Architectural Comparison

MA can be visually presented as in figure 1. The web service is a layered structure in which all the different layers have a specific task to perform. It follows the Model-View-Controller (MVC) design pattern (Reenskaug, 1979). The UI is the View, the business logic is the Controller, and the database is the Model.

The classical definition of MSA is the one given in the Fowler and Lewis (2014). A compatible definition for a single microservice can be found in Newman (2019).

Fowler and Lewis (2014) define the microservice style as follows: the service is componentized into smaller services so that each component is independently replaceable

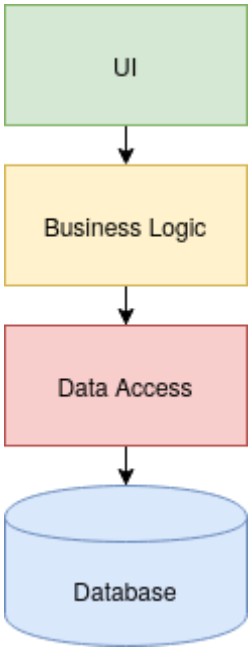

Figure 1: Traditional monolithic architecture (Kalske et al., 2018)

and upgradeable, the services are organized around business capability rather a design pattern (such as the previously mentioned MVC), a team should be responsible of their product for its full service life, the services are to contain the logic and communicate using a communication system without business logic, which can be summarized as "smart endpoints and dumb pipes", decentralized governance meaning that choices such as the technology to use or the architecture are not dictated to the developers, decentralized data management, infrastructure automation, fault tolerance of the whole application since individual services might fail or become unavailable, and evolutionary design.

Newman's (2019) definition for a microservice is a service that it is independently deployable, is modeled around business domain, owns the data that it needs to operate, communicates via a network, is technology agnostic, encapsulates data storage and retrieval, and has a stable interface.

An example of MSA, presented in figure 2, has many problem areas of which one is the challenging security implementation. This is because every microservice accessible to the client can also be accessed or contacted by other, more malicious parties in the same network. The network in the case of web services is the internet. The attack surface available for the malicious party is the entirety of the APIs offered by the microservices.

One solution to limit the attack surface is the addition of API Gateway to the architecture as in figure 3. Montesi and Weber (2016) present an API Gateway design pattern. In this pattern, there exists only one web service accessible to clients. The API Gateway is a natural place for a Policy Enforcement Point (PEP) and other more MSA-specific features such as service discovery. The security features can be implemented in the API

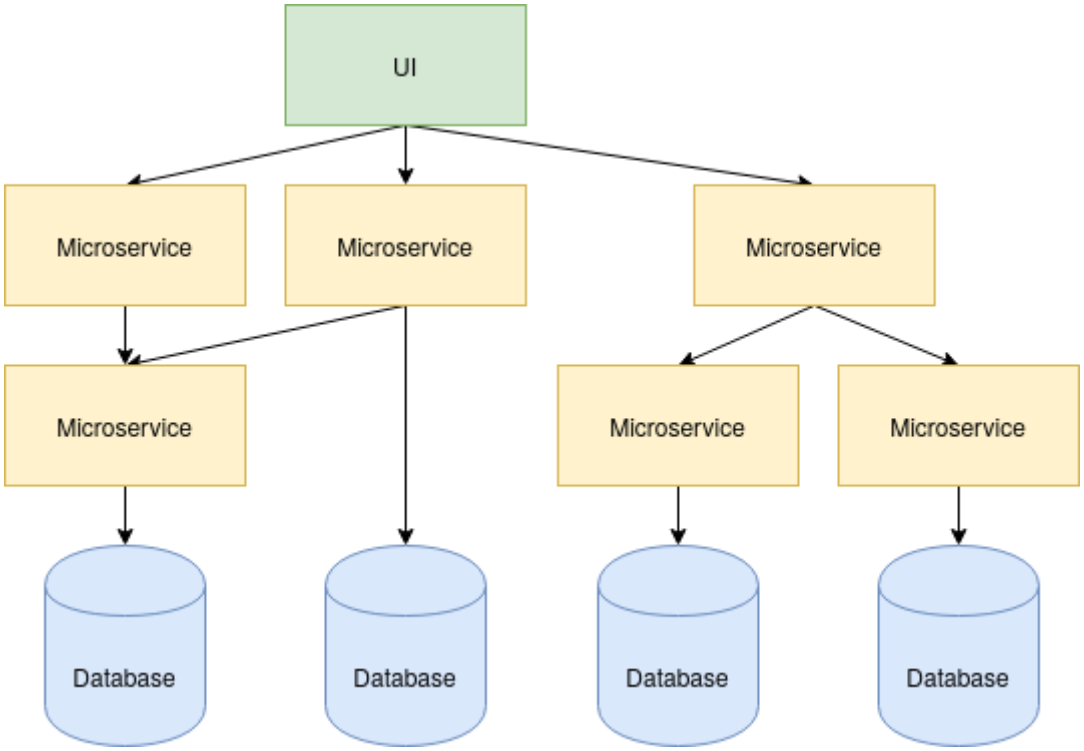

Figure 2: Microservice Architecture (Kalske et al., 2018)

Gateway, making it a critical component. Since all communication either flows through or is sanctioned by the API Gateway, its performance and accessibility are critical.

3 Changing the architecture

The changing of the architecture of an already deployed service from MA to MSA should be a gradual process. This ensures a smooth transition and minimizes outages to the customers. Sometimes though, this is not possible. Newman (2019) states that, when a monolithic application is implemented following a design patterns such as the MVC (Reenskaug, 1979), this can lead to difficulties in the refactoring. The reason is that the code base is not split according to the business domain but follows a rigid design pattern.

For the process to be as simple as possible, the MA is or at least should be split into modules with separation of concerns (Yarygina and Bagge, 2018). The splitting of the monolith can be carried out in various ways, one of which is Domain Driven Design (Evans, 2003). The selection of boundaries for the services is critical. If this is done incorrectly, all the affected services need to be refactored to mend the error (Newman, 2019).

MSA differs from MA in fundamental ways. According to Fowler and Lewis (2014), one main difference is the communication between the components. In a monolithic

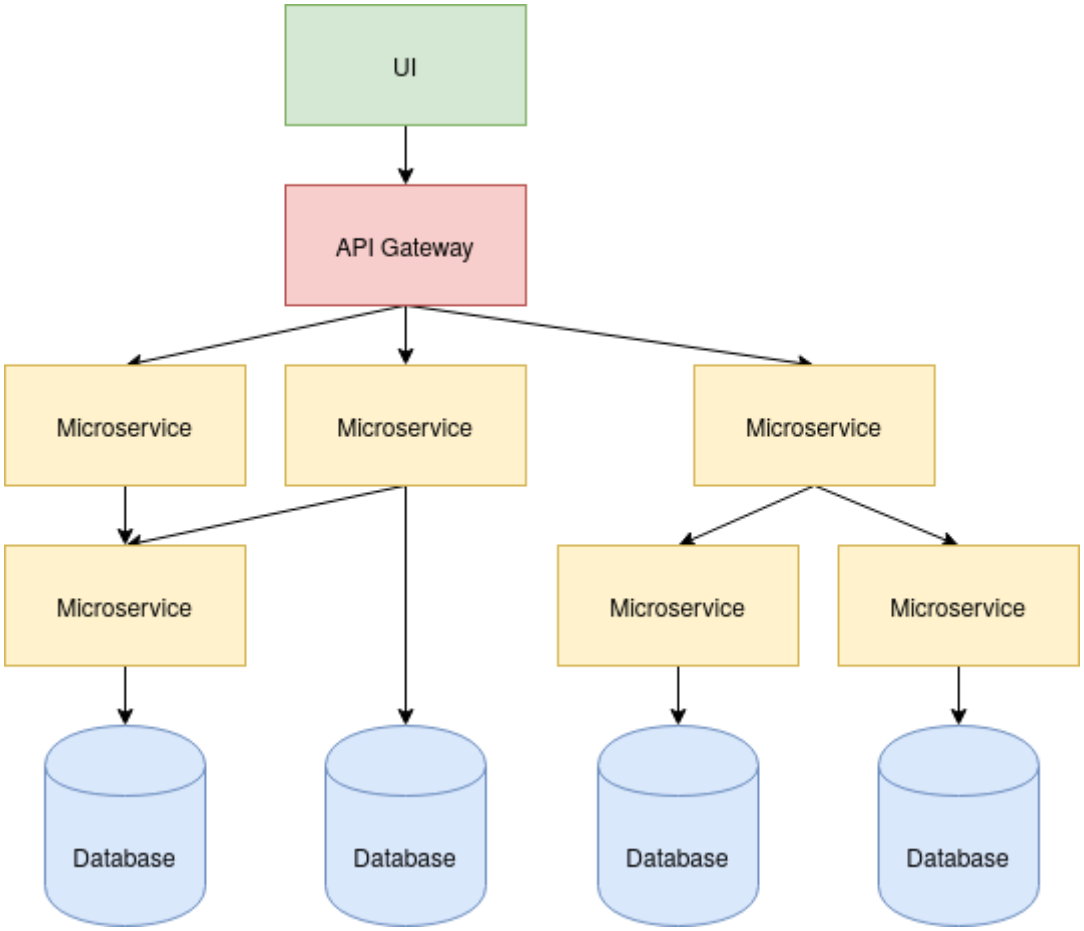

Figure 3: Microservice architecture with API Gateway (Montesi and Weber, 2016)

application, the processes can send function calls or method invocations among themselves. In MSA, the messaging is based on sending messages or HTTP requests over the network.

Communication using the network is slow, compared to local function calls. Function calls entail a stack frame creation in the call stack, execution of the function code, and finally popping the stack frame and returning the result. Compilers can optimize the code further and inline the function calls to eliminate the stack frame creation and the following procedures.

Requests sent to other microservices through a network are much slower than function calls within one computer. Therefore, the communication patterns should be changed to consider the change in the communication path. If the architecture is changed in such a way that the previous communication model among the components is preserved, there would be an excessive amount of communication and the resulting system would not be as performant (Fowler and Lewis, 2014).

Zari et al. (2001) studied the response times of web sites offered to the public. The website response times where measured in seconds. These times can be viewed as

being at the extreme end. Johansson (2019) did an experiment on a monolithic and a microservices application. The recorded response times for the monolithic application where, on average, 64% faster than for the microservice application.

As has been already established above the communication within the web service differs greatly in MA and in MSA. Secure communication is one of the key security aspects to be discussed in this paper. In addition to communication, the authentication and authorization are another critical security concern to be discussed. These will be addressed in the next chapter.

4 Authentication and Authorization

In the cases where the user must be authenticated, the web service needs a way to do this securely. There are many authentication schemes available, but users prefer the password (Zimmermann and Gerber, 2020). Due to this, the authentication is usually done with a pair of user credentials that contains a username and a password. The user is authenticated, and a key or token is transmitted to the user via the network. This communication in both MA and MSA should be encrypted in a way that none of the actors in the transfer path can intercept the message and misuse the credentials.

As has been mentioned previously, implementing security is hard and resource intensive. Therefore, the authentication implementation should adhere to an already existing framework or another entity providing the authentication. This is the case for both the MA and MSA. The available choices include Lightweight Directory Access Protocol (LDAP), OpenID Connect (OIDC), Security Assertion Markup Language (SAML), and Kerberos.

Regardless of the authentication scheme, the service or services authenticating the user contain the necessary information to do so. Since, the services in MSA are to be individually deployable, and a service should own its own data, implementing the authorization as part of the API Gateway, as a single micro service or using a third party authentication would be beneficial. The information would not be spread in to multiple services and multiple services would not have to have related implementation. In addition, whenever information is available, it is a target for thieves and hackers. Regardless of the implementation choices, the service must adhere to practices that minimize the risks of data breaches.

4.1 Authentication and Authorization in MA

In MA, it is possible to implement features in such ways that a process in which the application runs has access to a session or a user object that carries the user information. This information can consist of the granted roles and rights for the user. The information can be queried easily and securely when access control is needed.

An example of a web service implemented in MA is presented in figure 4. The user sends a request to a load balancer, which has all the information on the currently operational services. All the services run in the same process in which an authentication and authentication service or functionality is present. When a user first accesses the service, credentials are verified and, in a successful case, a session or a user object is created for the process.

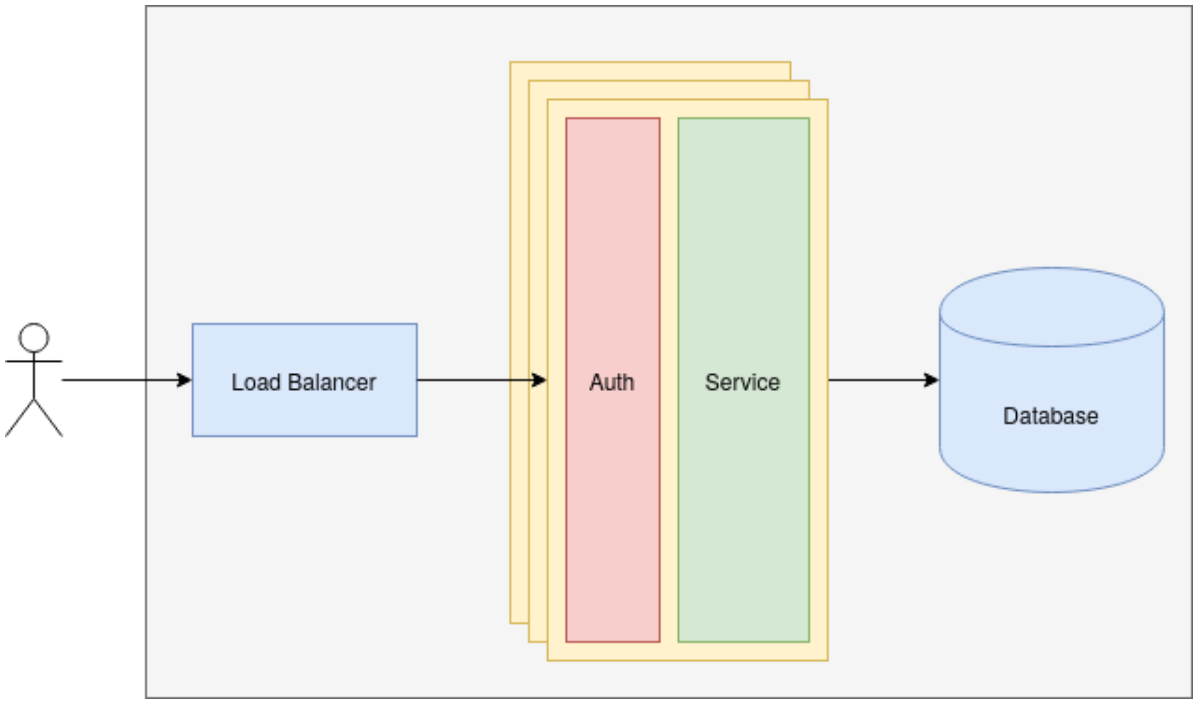

Figure 4: Traditional load balanced MA web service (He and Yang, 2017)

4.2 Authentication and Authorization in MSA

Authorization of the user rights can be implemented in various ways. One of them is an authorization service which can contain the access control list. Services verify from the authorization service that the client user or the role is allowed to access the requested service or functionality. The authorization is verified in memory and without any communication over the network.

In MSA, accessing the access control lists is not as easy as it is in MA. In order

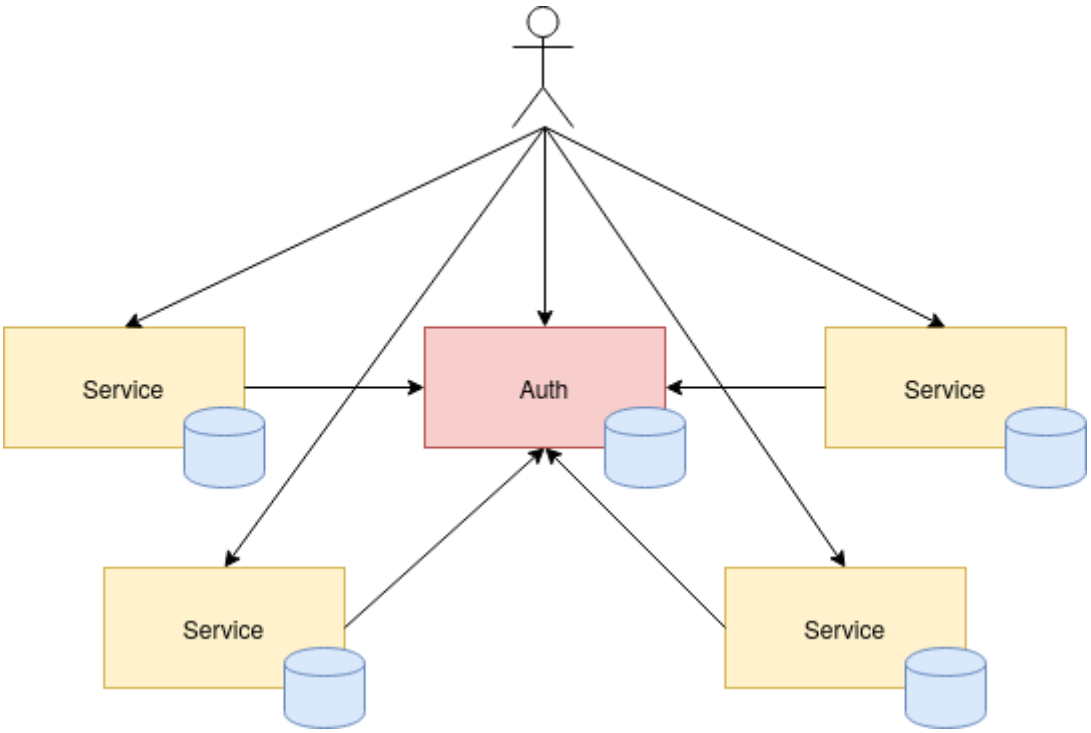

Figure 5: MSA authorization service (He and Yang, 2017)

to verify that a specific right exists, the service would have communication with the authorization service. This communication would need to happen every time a user tries to access a functionality with access restrictions. This could potentially lead to excessive communication from all the services and form a performance bottleneck at the authorization service. An example of such architecture is presented in figure 5.

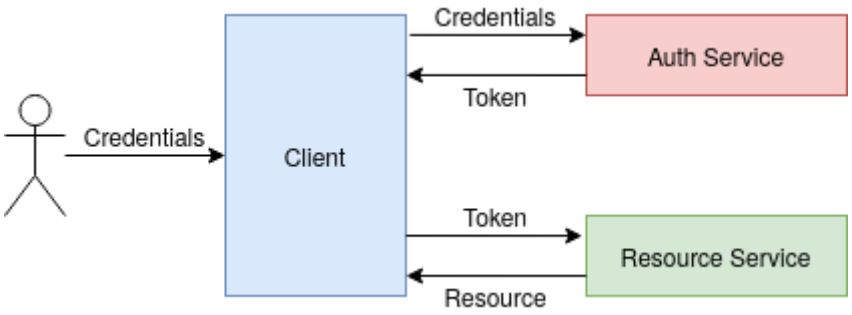

Figure 6: Bearer token-based authentication (He and Yang, 2017)

To limit the excessive communication, a token-based authentication and authorization scheme can be used. A simplified process flow is presented in 6. A user enters credentials to a login, and with these credentials, an authentication and authorization service grants the user a token. This token is used to access the services in the system. The services in question trust the issuer of the token and verify the token. If the token is accepted and access can be granted the request is serviced without any communication between

the service and the authorization service. A specific token-based authentication protocol is discussed in the next chapter.

4.3 OIDC

OIDC is an identity layer to accompany the OAuth 2.0 authorization framework-based protocol.

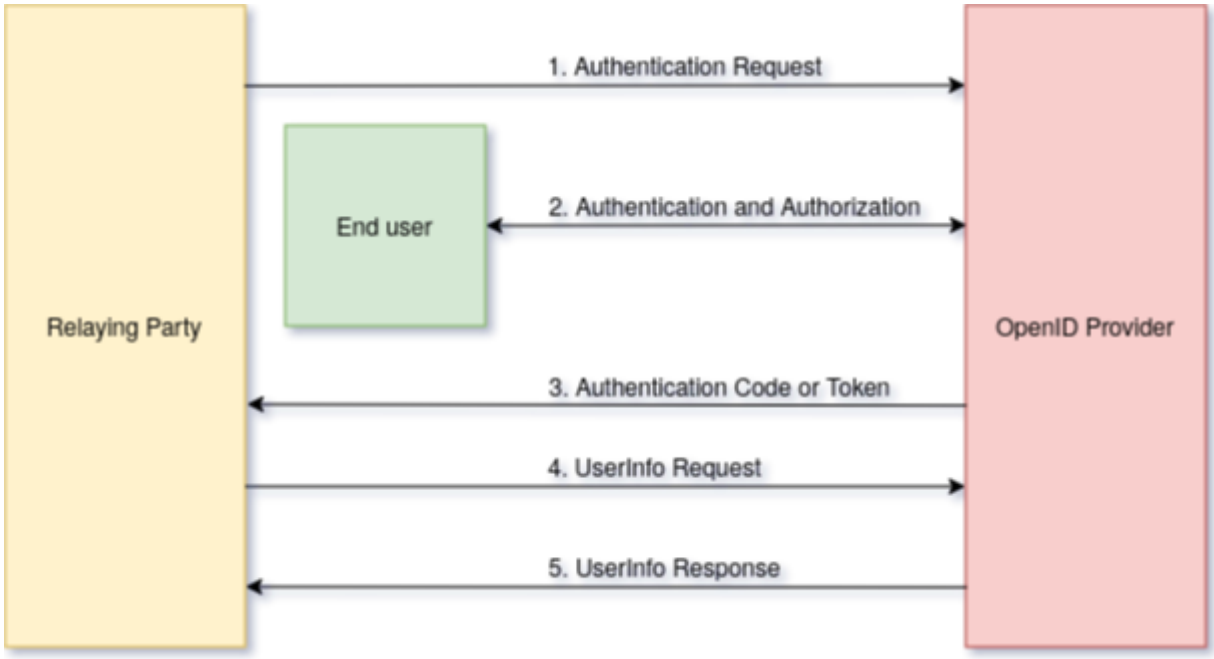

Figure 7: OIDC steps (oid, 2020)

An example of an OIDC flow is presented in figure 7. The process flow steps are the following: the client application requests an authentication from the identity provider, the end user is authenticated and authorization is obtained, the identity provider responds to the initial authentication request by sending an ID token and a possible access token, the client application can request the end user claims from the authorization server with the access token, and finally the authorization service returns the claims to the client application. The presented flow, is like the OAuth 2.0 flow, and the token sent to the client application is a JavaScript Object Notation Web Token (JWT).

4.4 JWT

JWT is a format to represent claims. It consists of base64 encoded, point separated strings, which can easily be carried in the HTTP request or response. The contents are key value pairs, and the token may or may not be signed and encrypted (Jones et al., 2015b). The token may contain an expiration time. If the token is used to validate

requests without a server-side implementation that can revoke a token, it will be valid until the expiration time. The JWT token must be issued by an authority trusted by the service.

The signing of JWT can be carried out in various ways. These are presented by Jones et al. (2015a). The signature is computed with the algorithm and keys or certificates specified in the header values. When the token is signed with a private key, it can be verified by all parties in possession of the corresponding public key.

4.5 Opaque Token

He and Yang (2017) compared several authentication and authorization solutions. One of the solutions discussed in the paper was the usage of an opaque access token for the client to API-Gateway communication, which is mapped to JWT in the API Gateway. The JWT would be used in all communication as the means to authenticate and authorize the user. The main problem the opaque token is said to solve is the logout problem. In comparison, a bearer token is valid until its expiration, and the client cannot invalidate the token. The opaque token flow is presented in figure 8.

The use of two tokens has many advantages. The token granted to the client allows access only to the API Gateway. This token cannot be used to access the services directly. Furthermore, the opaque token be can revoked by simply removing it from the storage. The internal JWT can alleviate some of the issues that arise when MA is to be changed to MSA. Firstly, token theft, though still possible, is limited to the time the client is logged on to the system. Secondly, though not advisable, the JWT can carry session information and the whole session. This allows for more of the previous MA implementation to be used without as many changes.

5 Communication

As already explained, the service components in MA can communicate using events, procedure calls or other methods available within a single server machine. Usually all this communication stays within a single computer and thus does not easily compromise confidentiality. This is not the case in MSA.

In MSA, the services communicate with each other via a network. There are multiple protocols and messaging systems to choose from such as, REST API, Advanced Message Queuing Protocol (AMQP), Enterprise Service Bus (ESB), and Remote Procedure Calls (RPC). For a more complete list view Yarygina and Bagge (2018). In this paper REST API will be examined further.

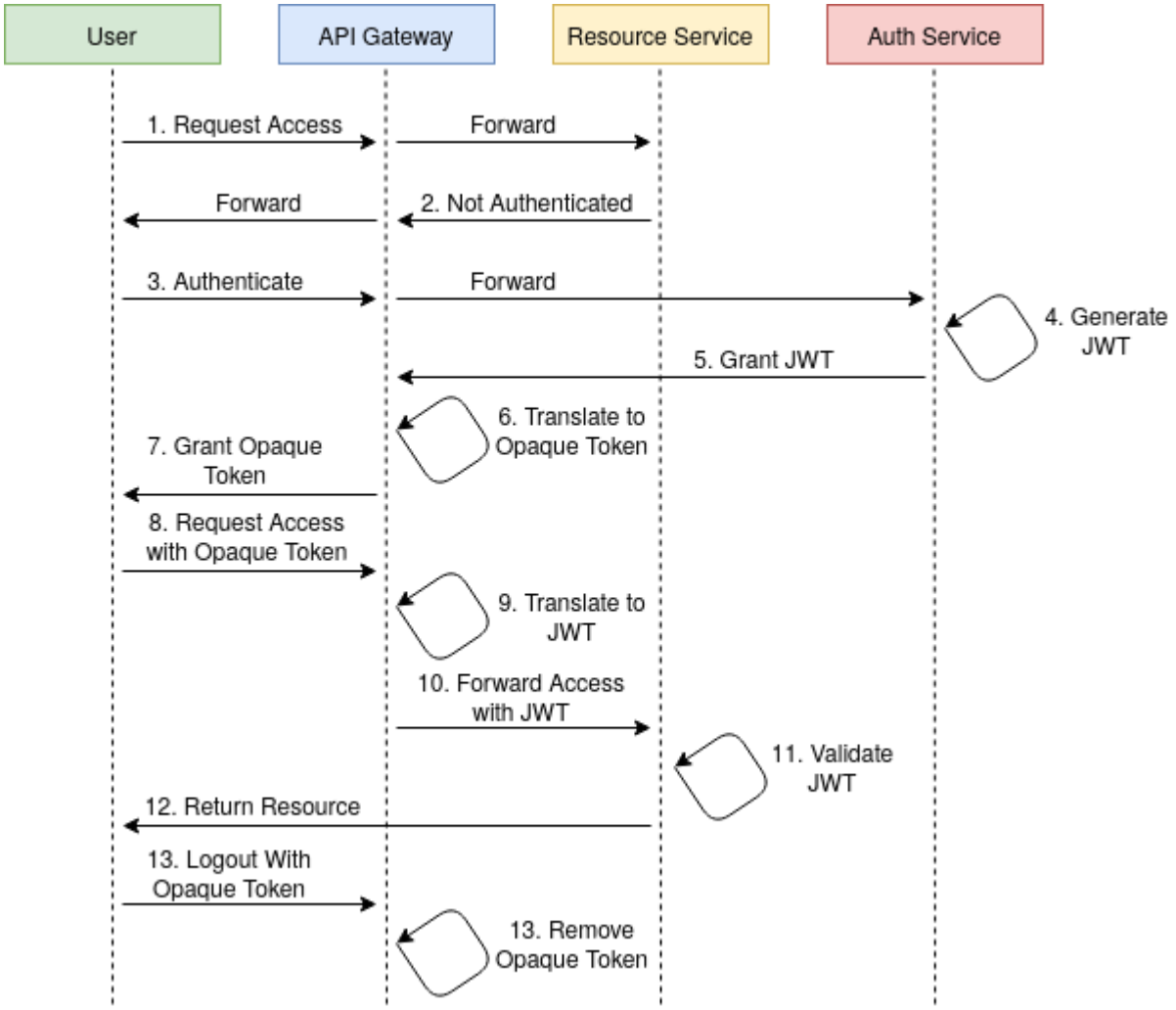

Figure 8: API Gateway and opaque token (He and Yang, 2017)

5.1 Representational State Transfer (REST)

Fielding (2000) presented REST in 2000. REST has become a very successful architectural style. The style was defined using various constraints, one of which is stateless communication. This entails that a request must contain all the information needed to fulfill the request because the server does not keep track of the client. All session state is stored in the client, of which the server has no prior knowledge before receiving the request. Yarygina and Bagge (2018) critique the REST paradigm from the security perspective. It states that the design of the architecture does not meet the security requirements for web applications. She also states that REST does not allow for any server-side sessions and thus token revocation is impossible. Tokens can be validated only for the correct issuer by signature and for expiration. As such tokens, are more compatible with REST, but there still must be the public keys in the server for the signature verification.

5.2 Coping With Failure in Communication

Montesi and Weber (2016) present widely used design pattern for MSA. The Circuit Breaker can be used to mitigate the case where a microservice operates slower than the services calling it and is unable to fulfill the requests. The circuit breaker is either implemented in the microservice or as a proxy between the client service and the microservice. When the microservice does not service requests as intended, the circuit breaker trips and sends a failure message to the clients immediately when requests are received, thus allowing the overloaded microservice time to service the prior requests.

The circuit breakers can prevent an application from becoming completely unresponsive and crashing when a denial of service attack is carried out on the service.

6 Deployment Automation and Production

Both MA and MSA web services can be installed on servers operated by the organization or individuals themselves. The software can be installed on the host operating system directly or a virtualization technology can be used.

Richter et al. (2018) implemented a test system mimicking the Deutsche Bahn seat reservation system using MSA. The purpose of the test system was to analyze security risks that were introduced by the implementation. In the study, they categorized the solution into three layers, the first of which was the compute provider, the second was the encapsulation technology, and the third one was the deployment. The technologies for these layers were: Amazon Web services, Docker for containers, and Kubernetes (k8s) nodes, respectively. They found out that the cloud-based infrastructure, when used in MSA, resulted in a more complex solution than in MA. The added layers such as the K8s, must be configured correctly, and an error in one could potentially compromise the whole system. In addition, the implementation of security is difficult and resource intensive. The rewards from good security, on the other hand are invisible. When microservices are implemented or even planned, the security should be considered as early as possible. Implementing security later in the project or as an afterthought can be more expensive and difficult.

6.1 Virtualization

Virtualization can be carried out in various ways but, in this thesis, it means virtual machines (VM) and containerization. VM is a complete installation of all the software needed for a system to run. In its basic form, a container uses the host operating system capabilities without the need to install an operating system or nonessential software

anew. Only the application and its dependencies are needed. VMs are run on the host by a hypervisor and containers by an engine or the host operating system. Running an application on a VM or in a container does not differ for the application or, in this case, for the web service, as the environment is similar.

Containers have considerably lower overhead when compared to VMs. A container can be created and started easily and automatically as needed by an orchestration solution such as Docker Compose, Docker Swarm or k8s. All these tools must be correctly configured and used according to their specification and best practices.

6.2 Orchestration

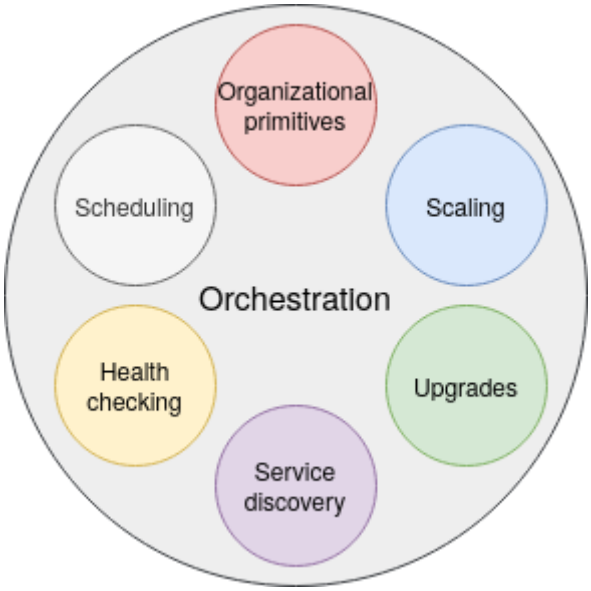

Figure 9: Orchestration constituents (Hausenblas, 2018)

Container orchestration can consist of the services and operations as is depicted in 9. Scaling refers to automatically creating or shutting down pods or containers to match the utilization level. The containers and the microservices within can have a new version that needs to be rolled into the production. The upgrade service is responsible in doing this. Service discovery is a service which is used to locate running services and as a service to which self-registration is carried out. If a service becomes nonoperational, it might not be able to send the orchestrator any message on this erroneous behavior. Therefore, the orchestrator can have a health check service. This service can periodically send a message to a service and, if no response is not received within a reasonable time, the service is deemed nonoperational and a new one is created by the scheduler. This service creates the individual pods or containers according to the system settings. The last orchestration part is the organizational primitives. These are used to e.g. label pods or containers with

matching business domain names. This is to make the administration and setup work easier (Hausenblas, 2018).

6.3 Service Mesh

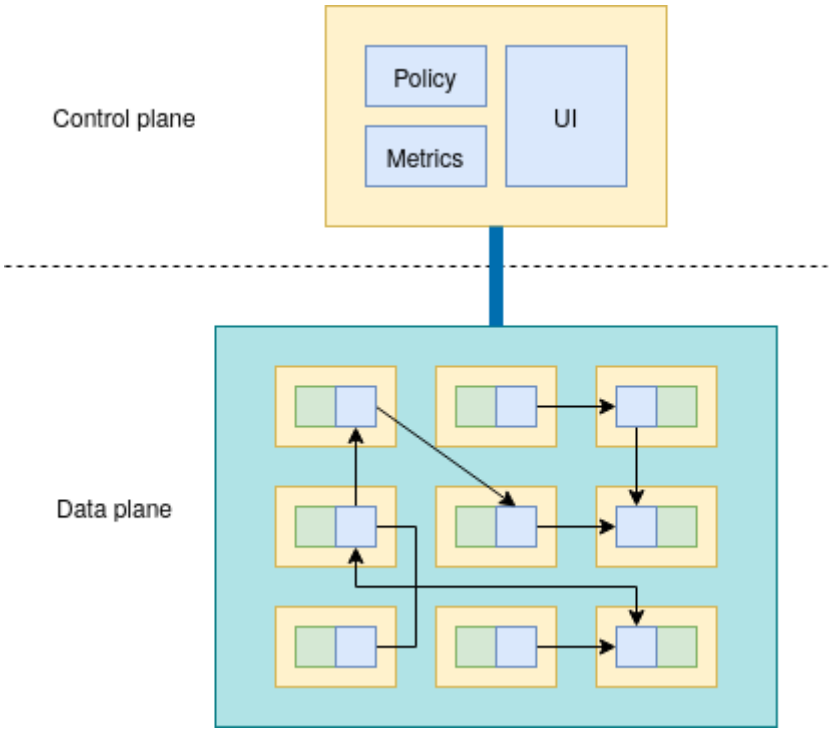

Figure 10: Service mesh basic architecture (Miranda, 2018)

A service mesh tries to solve the service-to-service communication challenges and to allow for monitoring of the entire system. In figure 10, a very basic service mesh architecture is presented. The architecture consists of a Control plane and a Data plane. The Control plane offers a user interface for system administrators, a Policy Information Point (PIP) and a Policy Decision Point (PDP), and collects metrics of the behavior and actions in the system. The data moves in the dataplane. Each microservice is accompanied by a proxy. All requests and responses flow through the proxies and are controlled by the Control plane (Miranda, 2018). The microservices do not have to be aware of the proxy that is in the information path.

There are several products available for a service mesh, such as Istio, Linkerd, and Consul Connect, to mention a few. As was the case with authentication and security in general, it is not advisable to implement security-critical features if there exists an off-the-shelf alternative.

The use of a service mesh greatly simplifies the microservice implementation since the proxy can contain many of the features that would otherwise be repeated in all the

services. The service mesh can provide a certificate authority for the communication between the services. The proxies act on behalf of the service, and all proxy-to-proxy communication can be encrypted on the transport layer with TLS (Calcote, 2019). Furthermore, it facilitates the use of mTLS where both parties are verified. A service mesh also allows for multi-cloud installations.

7 Other Security Concerns

7.1 Known Vulnerabilities in JWT

The choices for the algorithm for signing the JWT includes ”none”. This was found to be troublesome by McLean (2015). He found that many libraries did not operate in the desired way. The receiving party could be fooled to validate a mutated token without any signature with the ”none” as its algorithm. In addition to this vulnerability, McLean (2015) found that the verification suffered from another fatal flaw. A server can be fooled into to accepting invalid tokens. A malicious party can sign a JWT with the server public key and a symmetric algorithm. The token verification functions in some of the JWT libraries do not have the algorithm as one of the parameters and allow the algorithm field in the token to dictate the verification algorithm. If the keys match, as they should if the public key was correct, the library accepts the forged token as valid.

7.2 Transactions

The data in the system is the resource to be protected. If this data becomes corrupted, the system is not secure. In some cases, the actions to be taken consists of multiple reads and writes of the data, and the order of execution changes the result.

Transactions can be used when updating database content to make sure that atomicity, consistency, isolation, and durability (ACID) (Haerder and Reuter, 1983) are followed. When using MSA, according to the definition, each of the microservices should contain or have access to its own data i.e. a database. In MA, transactions are easier to implement, especially when the execution is done sequentially. In MSA, this is not necessarily the case. Performing an action might entail calling various microservices and if any of the individual actions fail permanently the changes that have already been made by other actions need to be undone. In addition to this, HTTP is asynchronous, and the execution order cannot be guaranteed.

Transaction-related issues can be solved by either splitting the monolith in such a way that the actions that need to be carried out as transactions are carried out in one microservice,

or by creating a service to coordinate the actions taken by the single microservices.

7.3 Software Development

When software is developed using MA, it is usually deployed as a whole and the program code can be compiled, tested, and used as a single unit or multiple modules. In contrast, a service implemented in MSA can be deployed in single-microservice units, and thus each component can be worked upon individually and deployed once ready.

The immediacy in the deployment of the microservices entails a very specific security risk. Ahmadvand et al. (2018) present threats from malicious insiders working on the services as developers or in other positions with access to sensitive information. In microservice development, the finished implementations are immediately released to production. There are steps in the CD pipeline prior to this, but once tests pass in the test environments, the pipeline is supposed to publish the changes to the actual production environment. The paper presents four specific threats. The first one is that the knowledge of sensitive information is spread among the developers more widely than in MA. This is due to access needs by developers. The second threat is that the insiders monitoring and operating the running system intentionally harm the system by making malicious changes. The third threat is the developers know the configurations and have the ability to make almost instantaneous changes to them or to the microservices themselves. The last presented threat in the paper is non-repudiation. The system is not able to disallow malicious requests when the developers have had access to the keys and other configurations. They can effectively implement services that emit malicious requests or responses. Malicious modification attempts in an MA are more easily screened by performing security audits and by peer reviewing the code. In an MSA, the knowledge of a single service and its inner workings are shared by a more limited number of people. Finding the compromised actions from the interoperation of the distinct microservices is a daunting task.

7.4 Externalized Configuration

To allow for easy configuration change management, there should exist a configuration orchestration service. This service should have an API from which services during their startup can load their appropriate configuration. The configuration of the whole system can be easily maintained through the API.

The contents of the configuration are highly sensitive information. It can consist of addresses, secrets, and other information that alter the behavior of the system. Secrets refer to credentials, connection strings, other keys and similar items that are to be kept confidential. Therefore, the content must be stored safely and not allowed to be read or

altered by unauthorized users.

7.5 Logging

Logging in MA is relatively easy. The chosen logging solution is used, and logs are created in easily configurable locations. In MSA, this can be more difficult. Each of the microservices needs to have its own logger, and each of these needs to be configured to log to a proper location. The logs that are created need to be persisted for the length of time allowed by legislation and according to the need of the system administration.

Without logs, it is impossible to verify correct operation of the system, nor is it possible to gain knowledge of a possible security breach.

7.6 Defense-in-Depth (DiD)

It is not enough to secure the boundary between the perimeter and the internal system. In DiD, the system should implement security measures at multiple layers within the system (Finnish Standards Association SFS, 2012). The system can be presented layered as in figure 11. Each of these layers should have security features, and a breach in one layer should not compromise the features of an inner layer in the system.

Gates (2019) takes the idea of layered defense even further and proposes an integrated approach to DiD. He defines the defense layers as edge routers, DDos defenses, Managed DNS, Reverse proxies, Bot Management, Web application firewalls, API defenses, and Caching. He further suggests that these layers or lines of defense should be aware of each other and be accessible from a single UI. Another approach he defines as human expertise. In this approach, there exists a command center that is always manned and, as such, able to respond to encountered threats. In some instances, this might be applicable, but for all web services it is not feasible.

8 Conclusion

This paper discussed the security aspects of changing the architecture from MA to MSA.

In practice, the previous MA that has been changed to MSA can have more aspects to go wrong than in the previous MA implementation. The deployment can entail installing, virtualization, monitoring, and other tools. In some cases, these tools must be implemented by in-house developers and, thus, more costs are incurred upfront and in the upkeep of the system. In addition to being more costly, the development has higher security risks involved.

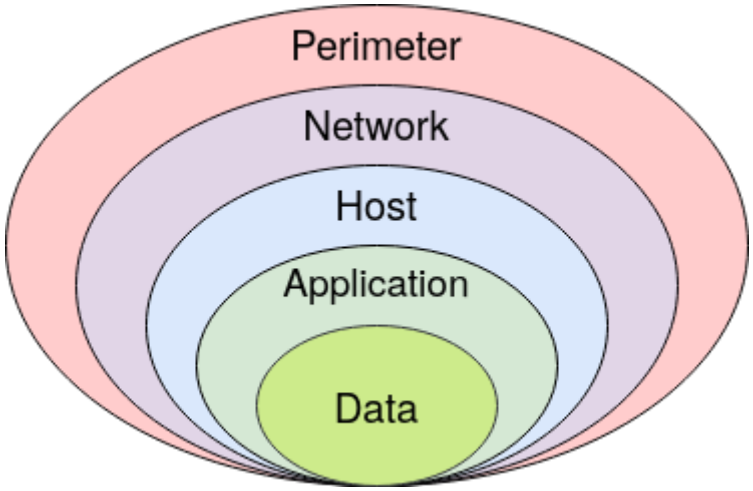

Figure 11: Defense-in-Depth

MSA has higher complexity due to more tools needed and having more potentially exposed attack surface. Security can be thought of as being only as good as its weakest link. In general, an MSA deployment has multiple layers, which all must be consistent and correct. One example is the configuration of the operating system on the server running the virtualization environment. All the layers from the server hardware to the handling of errors in the application code must be of high quality to prevent failures in security.

The communication that was in monolith a simple in-process call might not be possible as such in an MSA web service. The individual services communicate via the network with high overhead, in comparison to a simple function call. Furthermore, the identity and authorization of the entity requesting an action or data can usually be trusted in an in-process call. The mechanisms for proper authentication and authorization amount to even higher overhead for the MSA. There exists a very real risk for the development team to implement an insufficient security scheme. Bui et al. (2018) outline and implement a Man-in-the-machine attacks in which an insecure communication even within a single computer can be compromised. All network communication should be between verified counterparties and should use secure measures, even when this counterparty is running on the same server.

In MSA the security should be implemented in depth. There must be a healthy mistrust on all requests, and security should be built into the system. Security must be considered right from the beginning of the project in which the application architecture is transformed. The choices made in the development of the web service when following the MA do not carry to the MSA without changes.

The use of pre-existing tools and frameworks is highly encouraged. The tools currently available such as Docker, k8s, and Istio solve many of the inherent security issues if applied correctly. The tools and frameworks that exist cannot be blindly trusted. As

an example, some of the implementations using JWT had and in some cases still have serious flaws and are not secure.

Authentication and authorization in MA and in MSA can differ greatly. In MA, a session-based authentication and authorization is applicable. This is not the case in MSA, especially if a REST API is used. In REST API, all the information needed to serve a request should be enclosed in the request. Tokens such as JWT cater to this and can carry user information and other claims. The tokens are issued and signed by a trusted party. An API Gateway can act as a PEP and allow or disallow a request. Also, if a service mesh is used such as Istio, the enforcement can be done in the proxy for a specific microservice.

The results found in this study can be used to determine if an architecture change is feasible for a specific application.

In this thesis, many security aspects were not discussed. How do the many available design patterns for the splitting of the monolith fair when compared on security aspects? On the communication, only REST API on HTTP was discussed in any real extent.

Further research is needed on the security aspects of the different design patterns that are available for the architectural change. Also, the implementation of a field level access control should be studied further.

References

Welcome to OpenID Connect, 2020. Available `https://openid.net/specs/openid-connect-core-1_0.html`.

Mohsen Ahmadvand, Alexander Pretschner, Keith Ball and Daniel Eyring. *Integrity Protection Against Insiders in Microservice-Based Infrastructures: From Threats to a Security Framework*, pages 573–588. 06 2018. ISBN 978-3-030-04770-2. doi: 10.1007/978-3-030-04771-9_43.

Ross Anderson. Why information security is hard. *University of Cambridge Computer Laboratory, Tech. Rep*, 11 2001.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland and Dave Thomas. Agile Manifesto, 2001. Available `https://agilemanifesto.org/`.

Thanh Bui, Siddharth Prakash Rao, Markku Antikainen, Viswanathan Manihatty Bojan and Tuomas Aura. Man-in-the-machine: Exploiting ill-secured communication inside the computer. *Proceedings of the 27th USENIX Security Symposium, August 15–17, 2018, Baltimore, MD, USA*, 133:1511–1525, 08 2018. doi: 10.1016/j.ijhcs.2020.08.006.

Lee Calcote. *The Enterprise Path to Service Mesh Architectures*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2019. ISBN 9781492041795. 1st edition.

Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2003. ISBN 9780321125217. 1st edition.

Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine, 2000.

Finnish Standards Association SFS. Industrial communication networks. Network and system security. Part 1-1: Terminology, concepts and models. Standard IEC/TS 62443-1-1:fi, Finnish Standards Association SFS, Helsinki, 2012.

M. Fowler and J. Lewis. Microservices – a definition of this new architectural term, 2014. Available `https://martinfowler.com/articles/microservices.html`.

Stephen Gates. *Modern Defense in Depth.* O'Reilly, 2019. ISBN 9781492050360. 1st edition.

T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(5):287–317, December 1983.

Michael Hausenblas. *Container Networking.* O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2018. ISBN 9781492036845. 1st edition.

Xiuyu He and Xudong Yang. Authentication and authorization of end user in microservice architecture. *Journal of Physics: Conference Series*, 910:012060, 10 2017. doi: 10.1088/1742-6596/910/1/012060.

Gustav Johansson. *Investigating differences in response time and error rate between a monolithic and a microservice based architecture.* Ph.D. thesis, 2019. URL `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-264840`.

M. Jones, J. Bradley and N. Sakimura. JSON web signature (JWS). RFC 7515, RFC Editor, May 2015a. URL `http://www.rfc-editor.org/rfc/rfc7515.txt`. `http://www.rfc-editor.org/rfc/rfc7515.txt`.

M. Jones, J. Bradley and N. Sakimura. JSON web token (JWT). RFC 7519, RFC Editor, May 2015b. URL `http://www.rfc-editor.org/rfc/rfc7519.txt`. `http://www.rfc-editor.org/rfc/rfc7519.txt`.

Miika Kalske, Niko Mäkitalo and Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. *Current Trends in Web Engineering*, Irene Garrigós and Manuel Wimmer, editors, pages 32–47, Cham, 2018. Springer International Publishing. ISBN 978-3-319-74433-9.

Tim McLean. Critical vulnerabilities in JSON Web Token libraries, 2015. Available `https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/`.

George Miranda. *The Service Mesh.* O'Reilly, 2018. ISBN 9781492031321. 1st edition.

Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and API gateways in microservices. *CoRR*, abs/1609.05830, 2016. URL `http://arxiv.org/abs/1609.05830`.

Sam Newman. *Monolith to Microservices. Evolutionary patterns to transform your monolith.* O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2019. ISBN 9781492047841. 1st edition.

Trygve Reenskaug. THING-MODEL-VIEW-EDITOR an Example from a planningsystem, 1979. Available `http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf`.

Daniel Richter, Tim Neumann and Andreas Polze. Security considerations for microservice architectures. *Proceedings of the 8th International Conference on Cloud*

Computing and Services Science - Volume 1: CLOSER,, pages 608–615. INSTICC, SciTePress, 2018. ISBN 978-989-758-295-0. doi: 10.5220/0006791006080615.

Stack Overflow. Stack Overflow Developer Survey Results 2019. Available `https://insights.stackoverflow.com/survey/2019`.

Tetiana Yarygina and Anya Bagge. Overcoming security challenges in microservice architectures. pages 11–20, 03 2018. doi: 10.1109/SOSE.2018.00011.

Mazen Zari, Hossein Saiedian and Muhammad Naeem. Understanding and reducing web delays. *IEEE Computer*, 34:30–37, 2001.

Verena Zimmermann and Nina Gerber. The password is dead, long live the password – a laboratory study on user perceptions of authentication schemes. *International Journal of Human-Computer Studies*, 133:26–44, 08 2020. doi: 10.1016/j.ijhcs.2020.08.006.