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

Security in Microservice Architecture - Impact of a Switch from Monolith to Microservices

Bachelor's Thesis

17. huhtikuuta 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:	17. huhtikuuta 2020
Sivumäärä:	27
Pääaine: **Koodi:**	Computer Science SCI3027
Vastuuopettaja: **Työn ohjaaja(t):**	Professori Eero Hyvönen 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.

Tämän kandidaatintyön aiheena on tietoturva siirryttäessä monoliittisesta arkkitehtuurista mikropalveluarkkitehtuuriin verkkopalveluissa. Tämän kandidaatintyön 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 tunnistaminen ja valtuuttaminen.

Työssä havaittiin, että mikropalveluarkkitehtuuriinsiirtymisessä 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:	April 17, 2020
Pages:	27
Major: **Code:**	Computer Science SCI3027
Supervisor: **Instructor:**	Professor Eero Hyvönen Professor Tuomas Aura (Department of Computer Science)

There are many web services in use which were designed and implemented before the onslaught of microservices. These services might benefit from architectural change to microservices. Since the architectures are very different the security aspects need to be taken into account.

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

The research is carried out as a literary study. The literature consists of articles and textbooks. The articles used in this study were gathered from Scopus, GoogleScholar, and internet sources that were deemed reliable.

The main security considerations are: communication, configuration, and authentication and authorization.

Security has to be taken into account as early as possible in a project and the use of tools should be encouraged.

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

Contents

1 Introduction

In recent years, mobile applications and 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.

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 resources, a monolith might be a better fit to these companies trying to navigate to the actual product they are to offer. In the case of success, the need to rapidly scale 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 is furthermore amplified by the use of agile software development methods in which the change in requirements is welcome even in later stages of the development (Beck et al., 2001).

Kalske et al. (2018) finds 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 the needed and often mandatory changes to the code base.

New developers entering the workforce have a very different mindset than the older more seasoned professionals. Thus, it is very clear that the ways of working and paradigms to be 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.

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.

This thesis is organized as follows. The first chapter of this thesis compares the two architectures and the second chapter discusses the changing of architecture from MA to MSA. The third chapter presents the main security aspects of the change. The fourth and fifth chapter discuss the key security aspects that need to be addressed when the architecture is to be changed: access control, and system communication, respectively. In the sixth chapter other relevant security concerns are discussed. The last chapter in this thesis presents the conclusions and further research to be carried out on the subject.

2 Architectural Comparison

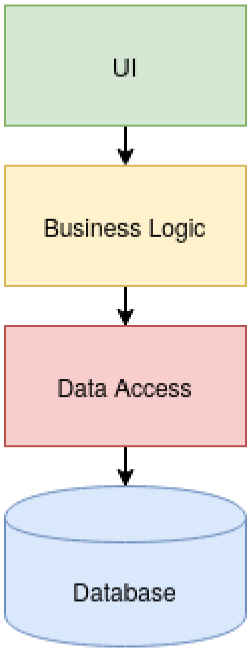

Figure 1: Traditional Monolithic Architecture (Kalske et al., 2018)

MA can be visually presented as in figure 1. The web service is a layered structure in which all of the different layers have a specific task to perform. This follows the Model-View-Controller (MVC) design pattern (Reenskaug, 2018). The UI is the View, the business logic is the Controller, and the database is the Model.

For an MSA the classical definition 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 to be componentized using services in which a component is independently replaceable 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 it's full service life, the services are to contain the logic and communicate using a communication system without business logic which can be simplified as "smart endpoints and dumb pipes", decentralized governance meaning that choices such as the technology to use or architecture are not dictated to developers, decentralized data management, infrastructure automation, the whole application needs to be fault tolerant since individual services might fail or become unavailable, and evolutionary design.

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.

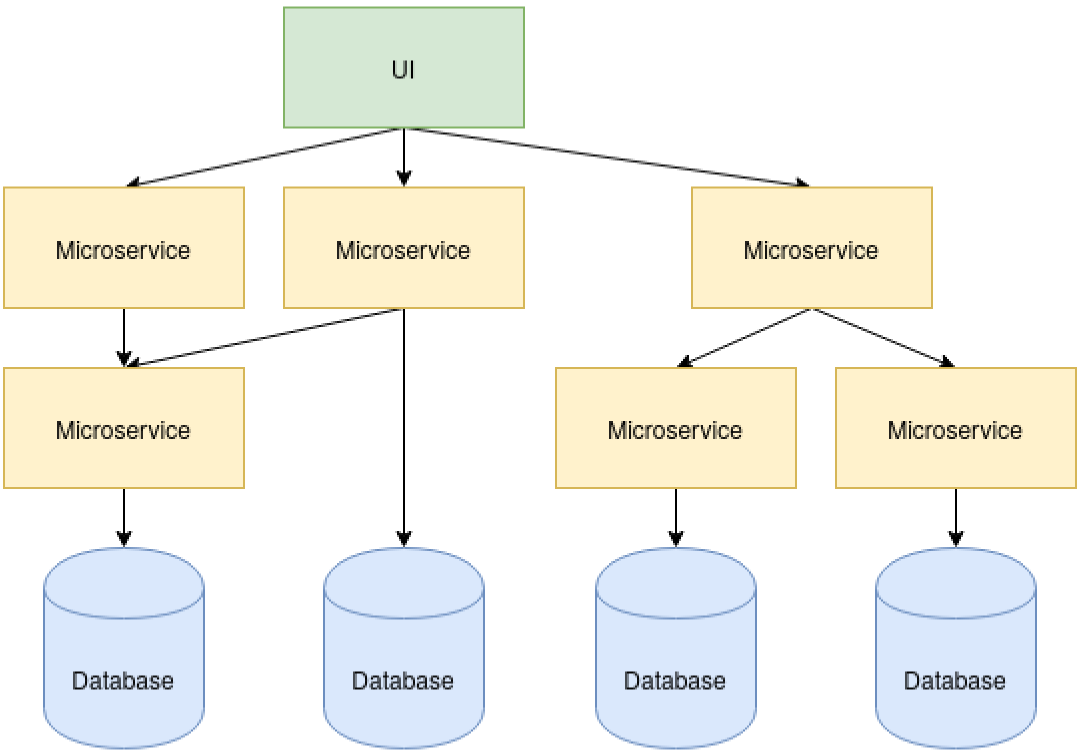

Figure 2: Microservice Architecture (Kalske et al., 2018)

An example for MSA, presented in figure 2, has many problem areas of which one is the challenging security implementation. This is due to the fact that 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 allows for 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 Gateway making it a critical component. Since all communication is to either flow through or be sanctioned by the API Gateway the performance and accessibility are critical.

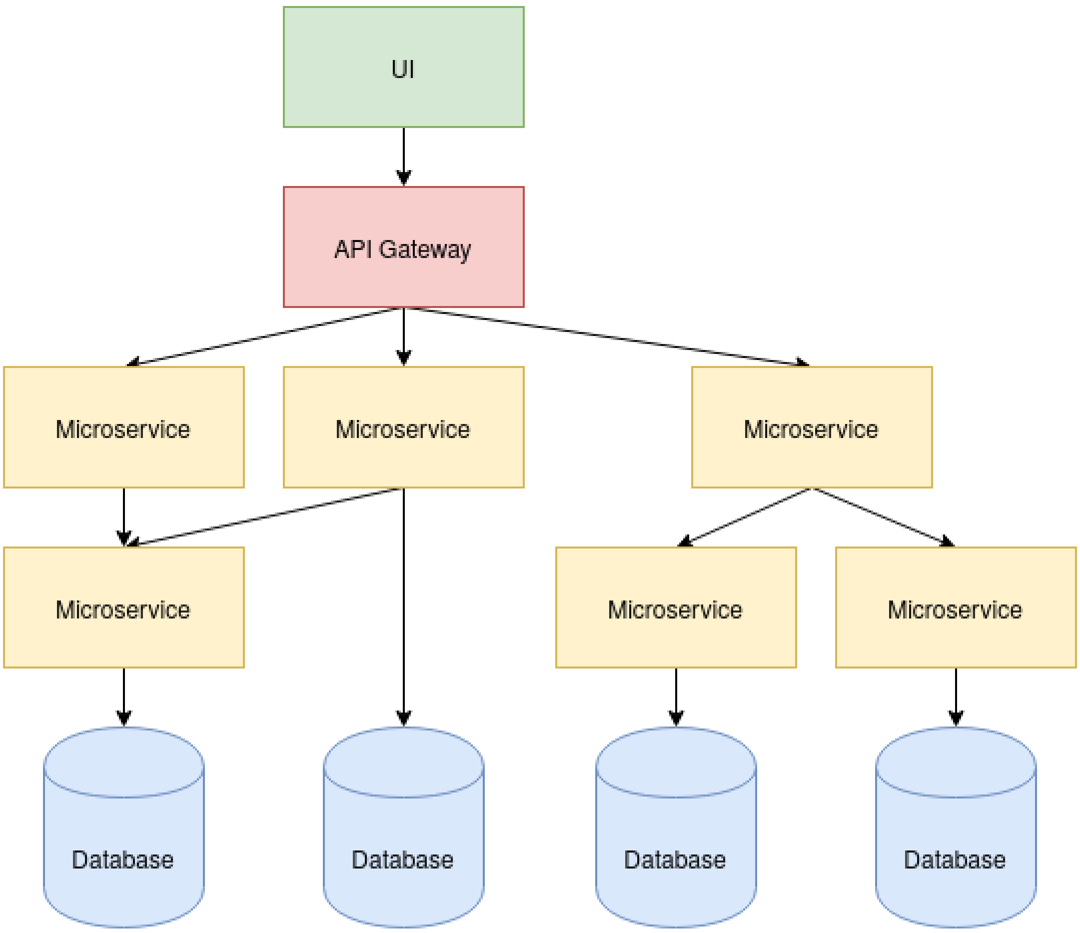

Figure 3: Microservice Architecture With API Gateway (Montesi and Weber, 2016)

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, 2018) this can lead to difficulties in the refactoring. Since the code base is not split according to the business domain but follows a rigid design pattern.

In order 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 actual 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).

The MSA differs from a MA in fundamental ways. According to Fowler and Lewis (2014), one main difference is the communication between the components. In a monolith application the processes can send function calls or method invocations among themselves. Where as in MSA, the messaging is based on sending messages or HTTP requests.

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

Zari et al. (2001) studied the response times of web sites offered to the public. The websites response times where measured in seconds. These times can be viewed as being on the extreme. 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 a microservice application.

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 take into account the change in 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).

4 Security

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 trying to defend a system. For the system to be secure the defending developers have to find all of the bugs where as the attacker has to find only one. Therefore, to make systems more secure the use of 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.

4.1 Identifying Key Security Aspects

As has been already established the communication within the web service differ greatly in MA and in MSA. Secure communication is one of the key security aspects to be discussed in this paper. In addition to the communication the Authentication and Authorization is the second critical security concern to be discussed. This will be addressed in the next chapter.

5 Authentication and Authorization

In the cases where the user has to 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, 2019). Due to this the authentication is usually done using a tuple containing user credentials i.e. 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.

The credential counterparts i.e. the secret shared by the server and the user have to be available for the web service for verification. When using MSA, the service should own it's own data. When ever information is available it is a target for thieves and hackers. The services in MSA are to be individually deployable and the service scalable. Authentication service implementation has to take this into account. The service has to adhere to practices that minimize the risks of data breaches.

As has been mentioned previously implementing security is hard and resource intensive. Therefore, the authentication implementation should adhere to an already existing framework or an 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.

5.1 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 or similar that carries user information. This information can consist of the granted roles and rights for the user. This information can be queried easily and securely when access control is needed. to execute an action or operation or a specific authorization service or module can exist.

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

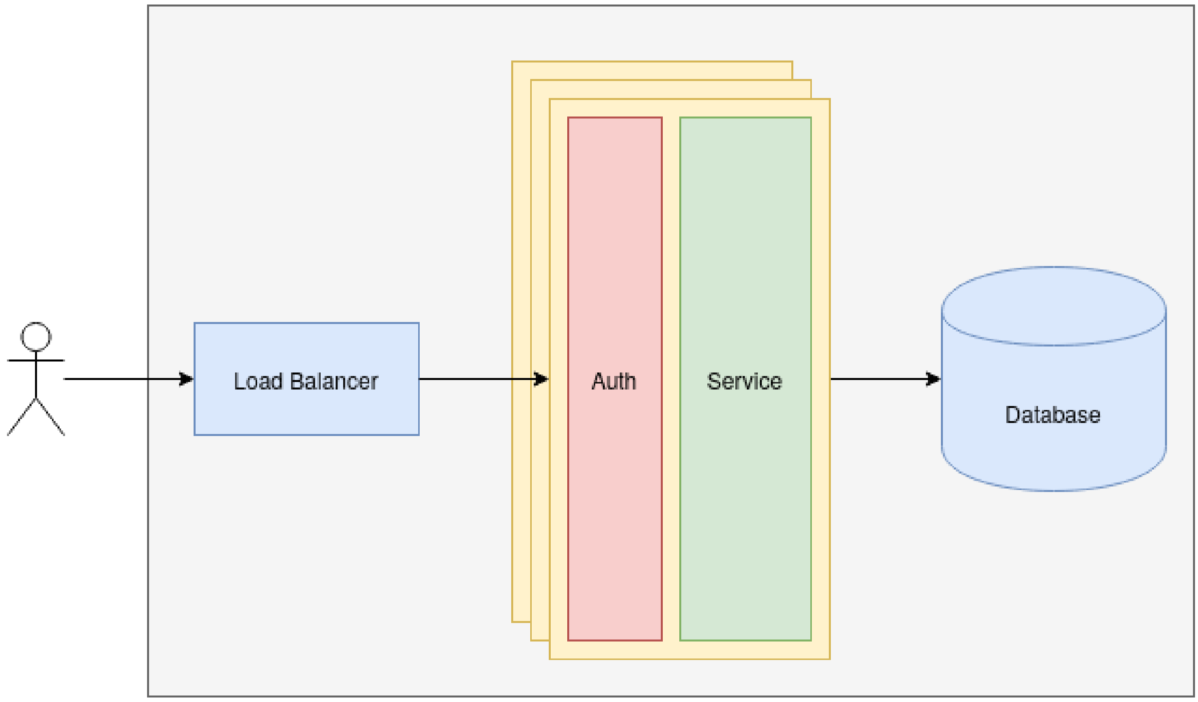

Figure 4: Traditional load balanced MA web service (He and Yang, 2017)

5.2 Authentication and Authorization in MSA

Authorization of the user rights can be implemented in various ways. One of which is an authorization service which can contain the access control matrix. Services being accessed verify from the authorization service that the client user or the role that the user has can access the requested service or functionality. In MA, the access rights to functionality can be implemented using annotations within the source code. The authorization is verified in memory and without any communication over the network.

In MSA accessing the access control matrix or matrices is not as easy as it is in MA. In order 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 extremely lively communication from all the services forming a bottleneck at to the authorization service. An example of such architecture is presented in figure 5.

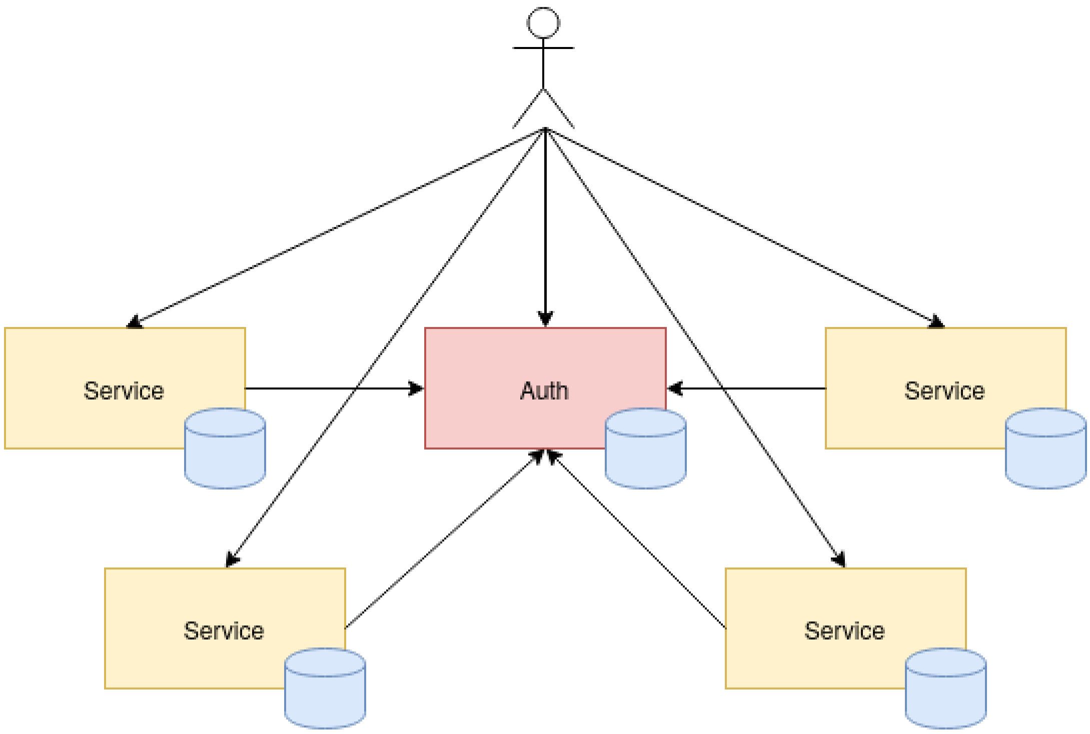

Figure 5: MSA authorization service (He and Yang, 2017)

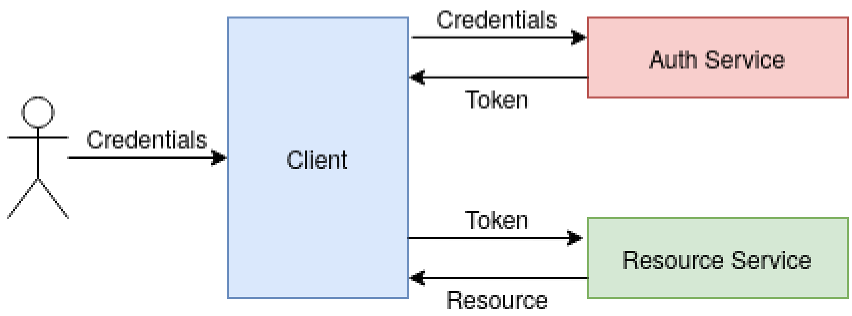

Figure 6: Bearer token based authentication (He and Yang, 2017)

To limit the excessive communication a token based authentication and authorization scheme can be used. A simplified process flow is presebted 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 and the claims within. If the token is accepted and access can be granted the request is serviced without any communication between the services and the authorization service. A specific token based authentication protocol is discussed in the next chapter.

5.3 OIDC

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

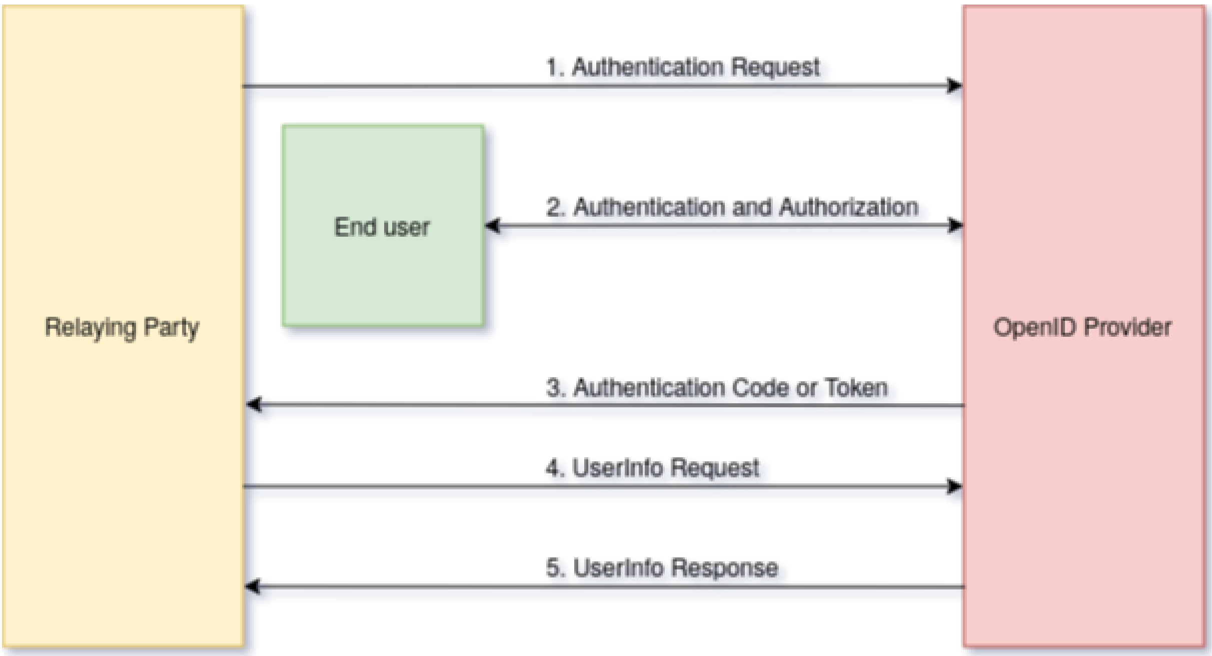

Figure 7: OIDC steps (oid, 2020)

An example of an OIDC flow is presented in figure 7. The process flow steps are: the 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 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 similar to the OAuth 2.0 flow and the token sent to the client application is a Java Script Object Notation Web Token (JWT).

5.4 JWT

JWT is a format to represent claims. It is base64 encoded, point separated strings, which can easily be carried in the HTTP request or response. The contents is 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 this time. The JWT token is issued by an authority trusted by the service. In figure a basic bearer token based authentication and authorization is depicted. A client initially enters credentials to a login page and the authorization service issues a token for the user. This token is used in place of the credentials to access restricted resources.

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

5.5 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 and API Gateway communication and map an Opaque token to JWT in the API Gateway. This would be used in all other communication as the means to authenticate and authorize the user. The main problem the opaque token is said to solve is the logout problem. A bearer token is valid until expiration and the client can not 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 can not be used to access the services directly. Furthermore, the opaque token be can revoked by simply removing it from the storage for the token pairs. The usage of an internal JWT can alleviate some of the issues 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.

6 Communication

As already explained, in MA, the service components 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 single services communicate via a network. There are multiple protocols or messaging system 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 is to be examined further. In few instances some of the other systems are discussed briefly.

Miranda (2018) lists the typical fallacies that developers fall victim to when designing

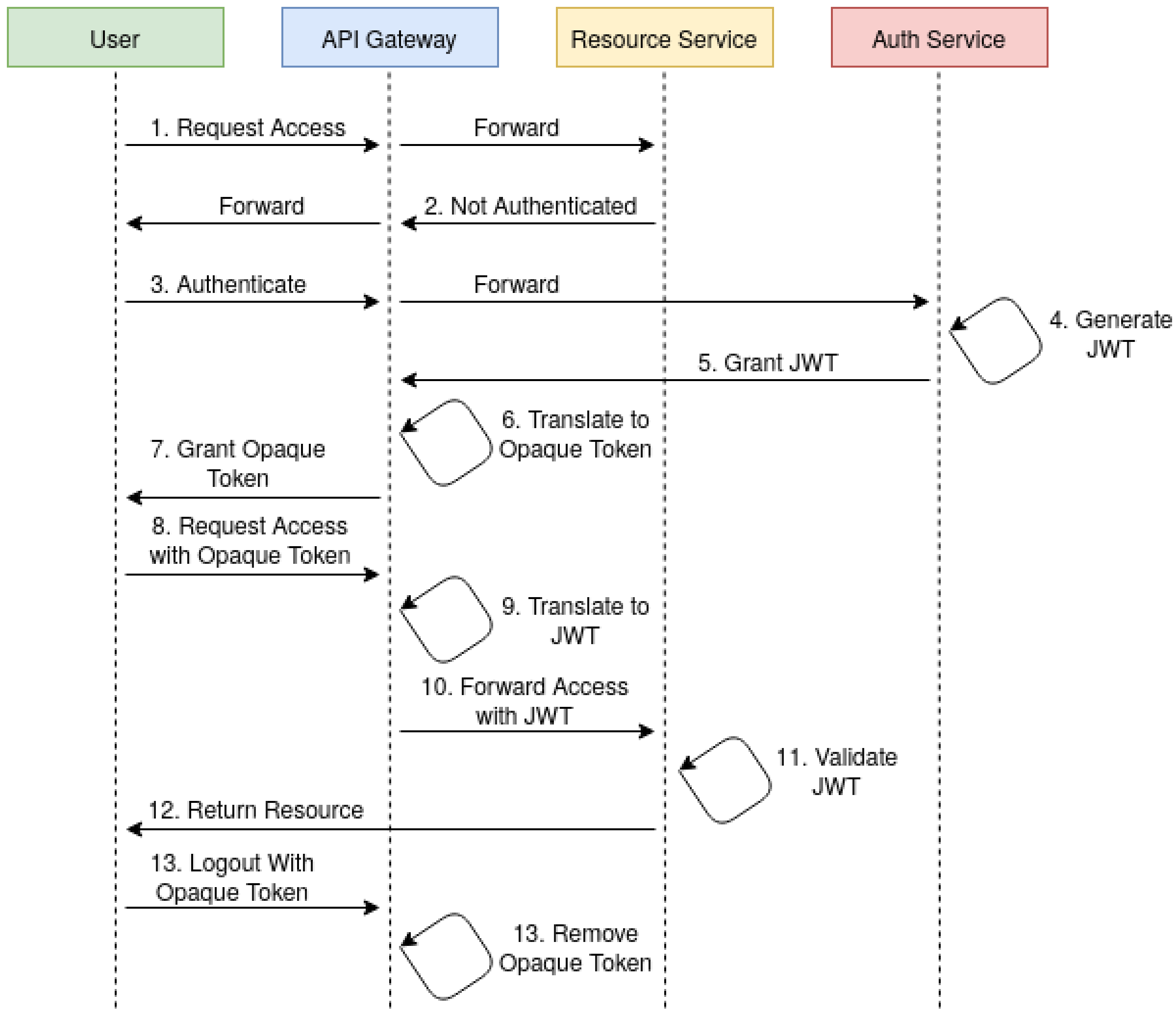

Figure 8: API Gateway and opaque token (He and Yang, 2017)

distributed systems:

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

6.1 Representational State Transfer (REST)

Fielding (2000) presented REST in 2000. REST has become a very successful architectural style. The style was derived 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 a request. In her doctoral thesis, Yarygina and Bagge (2018) critiques the REST paradigm from the security perspective. She 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 has to be the public keys in the server for signature verification.

6.2 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 very likely case that a microservice operates slower than the other services calling it and runs out of resources to fulfill the requests in time. The circuit breaker is either implemented in the microservice or as a proxy between the client 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 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.

7 Deployment Automation and Production

Both MA and MSA web services can be installed on servers operated by the organization or individuals them selves. 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 to three layers: 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, have to be configured correctly and an error in one could potentially compromise the whole system. In addition, implementation of security is very difficult and resource intensive. The rewards from a good security are invisible. When microservices are implemented or even planned the security should be taken into account as early as possible. Implementing security later in the project or as an after thought can be more expensive and very difficult.

7.1 Virtualization

Virtualization can be carried out in various ways but in this paper is to be on 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 operating system or non essential software a new. 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 VM or on a container does not differ for the application or in this case, for the web service, 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, and k8s. All of these tools have to be correctly configured and used according to their specification and best practice.

7.2 Orchestration

Container orchestration can consist of the services and operations as is depicted in 9. The scaling refers to automatically creating or shutting down pods or containers to match 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 is to become non operational 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 with in reasonable time frame the service is deemed non operational and a new one is to be created by the scheduler. This service is to create the individual pods or containers according to the system settings. The last orchestration part is the organizational primitives. These are to used to e.g. label pods or containers with matching business domain names. This is to make the

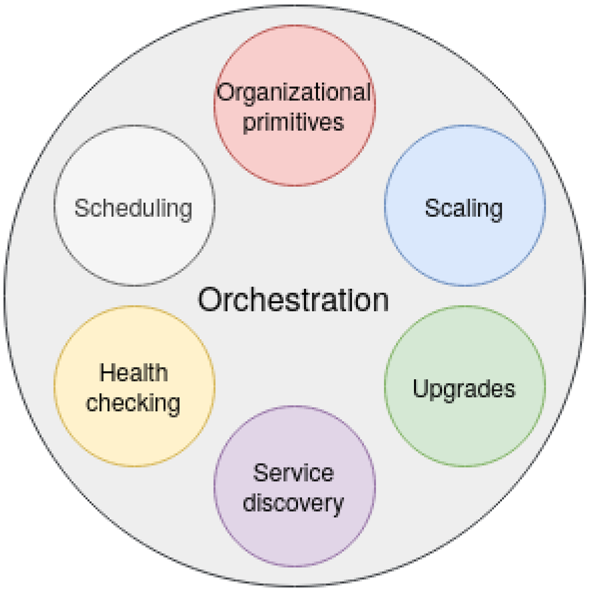

Figure 9: Orchestration constituents(Hausenblas, 2018)

administration and setup work easier (Hausenblas, 2018).

7.3 Service Mesh

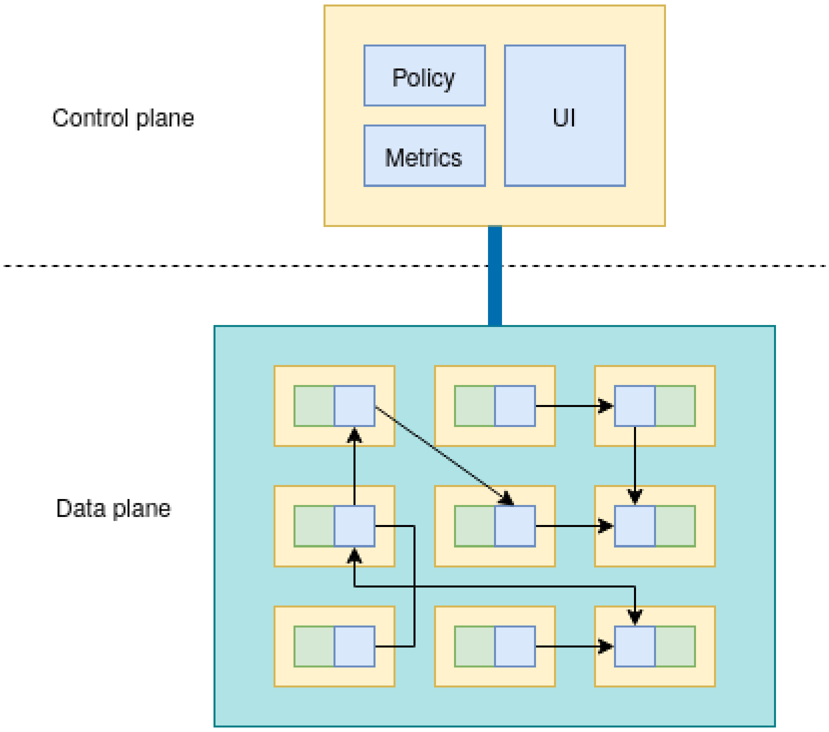

Figure 10: Service mesh basic architecture (Miranda, 2018)

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, Policy Information Point (PIP) and Policy Decision Point (PDP), and collected metrics of the behavior and actions in and of 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 service mesh greatly simplifies the microservice implementation since the proxy can contain many of the features that would otherwise repeat in all of the services. The service mesh can provide a certificate authority for the communication between services. The proxies act on behalf of the service and all proxy-to-proxy communication can be encrypted on the transport layer using TLS (Calcote, 2019). Furthermore, it facilitates the use of mTLS where both parties are verified. Service mesh also allows for multi cloud installations.

8 Other Security Concerns

8.1 Known Vulnerabilities in JWT

The choices for the algorithm for signing the JWT algorithm contain ”none” as one of the choices. This was found to be troublesome by . 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 it’s algorithm. In addition to this vulnerability McLean (2015) found that the verification suffered from another fatal flaw. When a token was created by using a symmetric algorithm, the servers could be fooled in to believing that a token signed by just the public key and not the secret HMAC key was a valid one.

8.2 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 end result.

Transactions can be used when updating database contents to make sure that atomicity, consistency, isolation, and durability (ACID) (Haerder and Reuter, 1983) is followed. When using MSA according to the definition each of the micro services should contain or have access to it's own data i.e. 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 permanentely the changes that have already been made need to be undone. In addition to this, a HTTP is asynchronious and the execution order can not 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 can be carried out in one microservice or by creating a service to coordinate the actions taken by the single microservices.

8.3 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 other positions with access to sensitive information. In microservice development, the finished implementations are to be 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 knowing the configurations and their ability to make almost instantaneous changes to them or the microservices themselves. The last presented threat in the paper is the 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 or requests that emit malicious requests or responds. Malicious attempts in a MA are more easily screened by performing security audits and by peer reviewing the code. In a MSA the knowledge of a single service and it's inner workings are shared by a more limited number of people. Finding the compromised actions from the interoperability of the distinct microservices is a daunting task.

Malicious attempts in a MA are more easily screened by performing security audits and by peer reviewing the code. In a MSA the knowledge of a single service and it's inner workings are shared by a more limited number of people. Finding the compromised actions from the interoperability of the distinct microservices is a daunting task.

8.4 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 in 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 is 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.

8.5 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 need to have their own logger and each of these need 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.

8.6 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 should not compromise those features on an inner layer of the system.

Gates (2019) takes the idea of layered defense even further and proposes an integrated approach to DiD. He defines the defence layers as: edge routers, DDos defenses, Managed

DNS, Reverse proxies, Bot Management, Web application firewalls, API defenses, and Caching. He further iterates that these layers or lines of defense should be aware of each other and be accessable from a single UI. Another approach he defines as human expertise. In this approach there exists an command center that is to be manned at all times and as such be able to respond to the encountered threats. In some instances this might be applicable, but for all web services this is not be feasible.

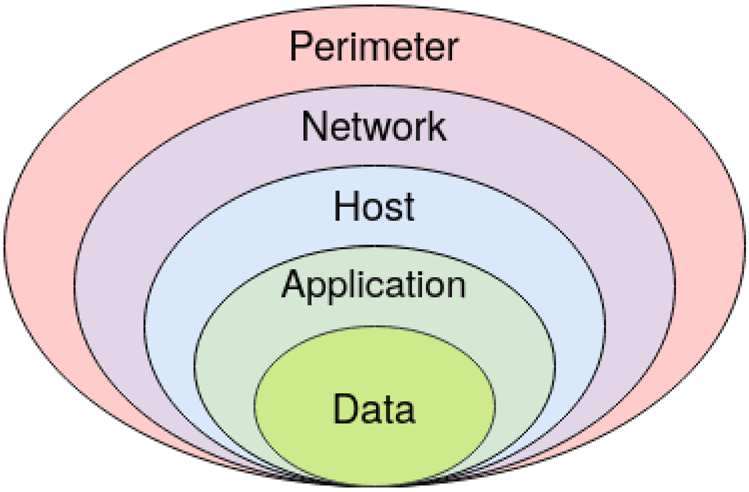

Figure 11: Defense-in-Depth

9 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 previous MA implementation. The deployment can entail installing, virtualization, monitoring, and other tools. In some cases, these tools have to implemented by in house developers and thus more costs are incurred upfront and also in the upkeep of the system. In addition to being more costly own development has higher security risks involved.

MSA has higher complexity due to more tools needed and having more potentially exposed attack surface. Security can be though of as being as good as its weakest link. In general, a MSA deployment has multiple layers which all have to be consistent and correct. One example is the configuration of the operating system on the server running the virtualization environment. All of the layers from the server hardware to the handling of errors in the actual code have to be of ample quality to mitigate a failure in security.

The communication that was in monolith a simple in-process call might not be possible as such in a 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 to allow 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.

In MSA the security should be implemented in depth. There must be a healthy mistrust on all requests and security should be built in to the system.

Security has to be taken into account right from the beginning of the project in which the architecture is to be changed. The choices made in the development of the web service when following a MA do not carry to the MSA as such.

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, can not 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 to be used. In REST API all the information needed to serve a request should be enclosed in the request. Tokens such as JWT cater 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 an PEP and allow or disallow a request. Also, if a service mesh is used such as Istio this can be done in the proxy for a specific microservice.

In this paper 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.

The results found in this study can be used to determine if at all architecture change is feasible in a particular case. This can be the case after other architectural changes or code refactoring. In addition, the results can be used as a guideline for future research.

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. Viewed 3.4.2020.

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. *Univesity 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/. Viewed 2.4.2020.

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, Geneva, CH, 2012.

M. Fowler and J. Lewis. Microservices - a definition of this new architectural term, 2014. Available https://martinfowler.com/articles/microservices.html. Viewed 15.2.2020.

Stephen Gates. *Modern Defense in Depth.* O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2019. ISBN 9781492050360. 1st edition.

T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *CM 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/`. Viewed 15.3.2020.

George Miranda. *The Service Mesh.* O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 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. MVC XEROX PARC 1978-79, 2018. Available `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`. Viewed 29.3.2020.

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`. Viewed 1.2.2020.

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, 08 2019. doi: 10.1016/j.ijhcs.2019.08.006.