Information Security

Security thoughts on modern software development

A survey about microservice security

Diploma Thesis

For attainment of the academic degree of

Diplom-Ingenieur/in

submitted by

Paul Lackner, BSc

1910619804

in the

University Course Information Security at St. Pölten University of Applied Sciences

The interior of this work has been composed in LaTeX.

Supervision

Advisor: Dipl.-Ing. Dipl.-Ing. Christoph Lang-Muhr, BSc

Assistance: -

St. Pölten, June 18, 2021

(Signature author) (Signature advisor)

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *Unterschrift*

Kurzfassung

Mit dem Aufkommen der Containertechnologie und Cloud Computing wurden Microservice Architekturen populär. Aber was sind Microservices? Warum wurden sie so beliebt und warum verändern sie die Funktionsweise von moderner Softwareentwicklung? Was sind die Vor- und Nachteile von Microservice, auch von Seiten der Sicherheit im Gegensatz zu monolithischen Services. Welche Herausforderungen gilt es zu bewältigen und welche Lösungen gibt es bereits dazu?
Die Microservice Entwicklung wuchs erstaunlich schnell seit ihrer Entstehung im Jahr 2007, trotzdem gibt es seither noch keine Richtlinen oder Best Practises zu einer sicheren Entwicklungsweise. Dies verunmöglicht es kleinen Unternehmen sichere Microservices zu programmieren, da die Ressourcen für die Auffindung von Problemen und Lösungen fehlen; selbst Unternehmen mit großem Umsatz arbeiten großteils nach Gutdünken und arbeiten nach keinem Standard, weil schlichtweg kein Standard existiert.
Diese Arbeit sammelt Eigenschaften von monolithischen Services und Microservices und sammelt dazugehörige Sicherheitsherausforderungen, die durch die Microservice-Architektur entstehen. Bestehende Lösungen zu den Problemen werden präsentiert und diskutiert, ob diese auch sinnvoll sind. Dabei unterscheidet diese Arbeit zwischen allgemeinen Konzepten und bereits konkreten Implementationen.
Die gesammelten Ergebnisse werden in einem Entscheidungsbaum zusammengefasst, der die Entscheidung für oder gegen die Nutzung von Microservices im konkreten Fall erleichtern soll. Wenn Microservices erstellt werden sollen, gibt es dann noch eine Checkliste, was bei der Erstellung von Microservices sicherheitstechnisch beachtet werden muss.

Abstract

With the advent of container solutions and cloud computing, micro services are becoming more popular. But what are micro services? Why are they becoming so popular and why are they changing the mechanics of modern software development? What are their advantages and disadvantages, also in terms of security, in comparison to monolithic services? What are the challenges to overcome microservice security and what are the existing solutions to them?

Microservice development has increased incredibly since its beginning in 2007, but still no complete guideline to a secure development exists, which makes in nearly impossible to small companies to really create trustful and secure services; even high budget companies do basically what they feel like, and share no standard as there is none.

This work collects properties of monolithic services and microservices and collects security challenges that arise when using a microservice approach. Existing solutions to these challenges are presented and discussed, if they are applicable. This work distinguishes between general concepts and concrete implementations.

The collection results in an decision tree, whether or not to use microservices and if so, a checklist what security aspects needs to be considered when using them.

Contents

1 Introduction

Flexibility. The business wants to maximise flexibility, as it removes the necessity of determination. The flexibility capacities of companies rises with microservices combined with cloud computing, but when new technologies arise, the security part is generally seen as an annoying annex which often seems to be forgotten. Also, security is often a quite simple thing to implement, yet it needs some brainpower to invent security controls and find already existing ones, that are appropriate for a project. The motivation of this work is to give an overview of what has already been done, and why these things matter. Furthermore, when creating and working on projects, what needs to be regarded and why. Security should not be a thing only gifted people deserve but needs to be understandable and simple enough for everybody to implement in the right way. This work aims to be a guideline on what to implement, why to implement and how to implement.

The relevance of this work is clearly visible when looking at some numbers. The microservice design approach, inheriting some aspects of the Service Oriented Architecture (SOA) architecture, dates back to the year 2007. Since then, the company Docker increased popularity in container technology, and its side products. It also got a huge market push with the wide adoption of the cloud, resulting in a cloud designed to host container and automate the orchestration via their own management tool [1]. This single cloud had a revenue of around 13 billion dollars[2] which does not include any other cloud revenues. This number clarifies the market relevance of containers and microservices. A recent study shows that a majority of companies began using microservice during the last two years[3], making it a recent topic. The US government has spent around 18 billion dollars in 2020 to protect against cyber attacks[4], while the average damage done by these attacks to companies varies from 24.000 to 504.000 dollars, depending on the company size[5]. Attacks on

[1]`https://cloud.google.com/`

[2]`https://www.cnbc.com/2021/02/02/google-cloud-lost-5point61-billion-on-13point06-billion-revenue-last-year.html`

[3]`https://www.oreilly.com/radar/microservices-adoption-in-2020/`

[4]`https://dailycaller.com/2021/01/08/solar-winds-hack-cyber-attacks-pentagon-homeland-security-budget/`

[5]`https://www.statista.com/statistics/1008112/european-north-american-firms-cyberattack-cost/`

infrastructure not only damage the budget of organisations but also cost lives[6]. Showing these numbers visualises the need to secure infrastructure and the increasing number of microservices.
Microservices raise a new series of security issues, which were not present in traditional software design. As microservices are individual services working together, the problem of shared infrastructure and intensive data sharing moves to a next level. Microservices make use of container technology which has a similar impact on the IT industry like the wide adoption of virtualisation and hypervisor and the shift to cloud service infrastructures. Microservices summarise the problems of containerisation, virtualisation and cloud computing. This work aims to visualise most of the problems and provides solutions for the individual challenges. The resulting research questions of this work are:

- What are the current security challenges in the microservice approach?
- What is different to those in the monolithic approach?
- What needs to be regarded when creating a new service resp. when migrating a monolithic service to a microservice?
- What are the current solutions to the relating security challenges?

1.1 Thesis Outline

This document is organized in several parts. In chapter 1, the topic is introduced, as well as problems, challenges and motivation of the work. It also clarifies the research question of the work. In chapter 2 some prerequisites and fundamental knowledge are described. The terminology of related services is described, and also a basic property evaluation of microservices takes place. Next, the overall trends and challenges of microservices are described. The last section describes basic security concepts which are necessary to understand for further understanding of microservice security. Related work is listed and briefly described in chapter 3. In chapter 4, security challenges of microservices are discussed. A layered approach of classifying security-relevant issues is introduced. Furthermore, the various types of security measures are described. A major part in this work are the Security Thoughts and Recommendations on Microservices. They describe general problems of microservices regarding security and how to deal with them, followed by section 4.4 which describes implementation-ready approaches. A short analysis of privacy issues concerning microservices follows. The chapter is finalised with a checklist, what needs to be done and remembered when creating and maintaining microservice structures. As every paper, this one also concludes in the last chapter 5, followed by the annex.

[6]https://techhq.com/2020/09/when-cyberattacks-cause-more-than-just-digital-damage/

2 Prerequisites

This paper discusses microservices compared to monolithic services in terms of security. The basics of the matter, the terminology and a basic evaluation of the properties of microservices, as well as relevant basic concepts of security are explained.

2.1 Terminology

In order to follow the topic and have a clear understanding of it, the definition of the various terms will be described.

2.1.1 Service Oriented Architecture

Service Oriented Architecture (SOA) is defined by Josuttis: "SOA is an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners" [1]. It has three core concepts [2]:

- *Service* is "an IT realization of some self-contained business functionality" [1]. They have the following attributes: self-contained, coarse-grained, visible/discoverable, stateless, idempotent, reusable, composable and vendor-diverse.
- "SOA *interoperability* in most cases is achieved via the Enterprise Service Bus that enables service consumers to call the service providers" [2].
- *Loose coupling* ensures the easy maintenance by SOA. Its principle is to minimize dependencies. A change in one service does not require a change in another service. Zimmermann states that previous approaches to SOA computing and microservices do not differ in architectural style [3].

Also, SOA is a further approach to separation of concerns. Separation of concerns means to split a complex problem into smaller ones and is one of the basic fundamentals of software design [4]. A program which is sticks strictly to this principle is called a modular program. In the SOA approach concerns are separated into services.

2.1.2 Monolithic Service

Traditional software is mostly created as a monolithic service. A monolithic software is written and compiled as one big project, where all parts play along and cannot be split from the core application. There is no external view into the logic of a running program (except for Reverse-Engineering and debugging tools). Function calls happen in the inner part of the software and there is one memory management for this software. As seen in Figure 2.1, a monolithic application includes a frontend and a backend in the same application. Each type includes functions which can only be triggered by the program itself, except some functions on the frontend which are explicitly designed for a user- or program- interaction.. Although monolithic services may be built in a modular way and has parts that can be distributed [2], it still remains a single executable artifact [5]. All the business logic, data access functions and the user interface are written in one application.

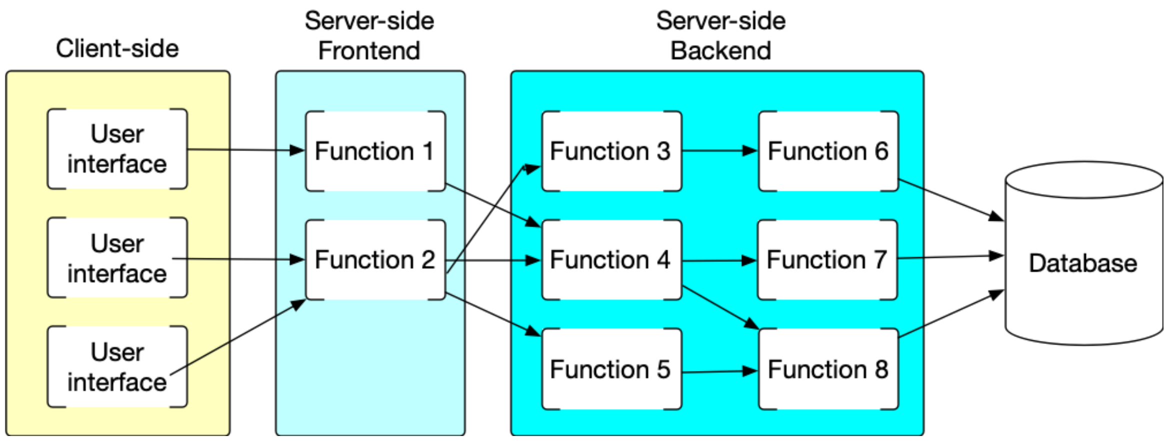

Figure 2.1: A monolithic application sketched as described by Levcovitz *et al.* [6].

2.1.3 Microservice

Microservices are a completely different approach on writing software. Instead of creating one big program, many little programs that work together are written [7] [8]. To communicate, microservices use common methods like REST Application Programming Interface (API)s. These small services have some properties that differentiate them from traditional programs. Pahl *et al.* describes microservices as “independently deployable, usually supported by a deployment and orchestration frame-work, e.g., in the cloud” [7]. Usually, to stay light-weight, microservices are deployed in a container, a light-weight virtualisation tool. Yale Yu *et al.* describe microservices as follows: “A microservice is an application on its own to perform the functions required. It evolves independently and can choose its own architecture, technology, platform, and can be managed, deployed and scaled independently [...] with its own release lifecycle and development methodology” [9]. Another definition of microservices is found on the website *microservices.io*. According to this definition, microservices are[1]:

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities
- Owned by a small team

Another good summary of the term *microservices* is written by Newman: “model [services] around business concepts, adopt a culture of automation, hide internal implementation details, de-centralize all things, isolate failure, and make services independently deployable and highly observable“ [10]. Microservices match the UNIX philosophy “do one thing and do it well”[2]. Figure 2.2 shows a schematic microservice architecture. A user interface calls a service which can call various other services in succession. There are multiple data access services with multiple databases or filesystems in this architecture. Also, the services are not separated into a front-end and a back-end. The user interface usually runs on a client (which can be seen as a distribution), while the services usually run on a single or multiple server. There are multiple opinions, which role microservices play in software architecture [2]. Microservices fit well in modern software engineering as Agile Development and Domain Driven Design (DDD). They marvelously exploit the advantages of containerization and cloud infrastructure. Some say, microservices are from their own architecture style [11], some say they are SOA [10] [3], while others say they are a redefined SOA [5] [12]. “Microservices can be considered meta-processes in a Meta Operating System (OS)” [13]. There are two types of microservices; microservices only communicating to other internal services, and microservices communicating to internal and external services [14]. Services only communicating to external services are defined as monolithic services. Internal services lie within the system boundary. Microservice instances are referred to as units.

To summarise this, microservices are small stand-alone applications which create a bigger application when they interact.

[1]https://microservices.io
[2]https://en.wikipedia.org/wiki/Unix_philosophy

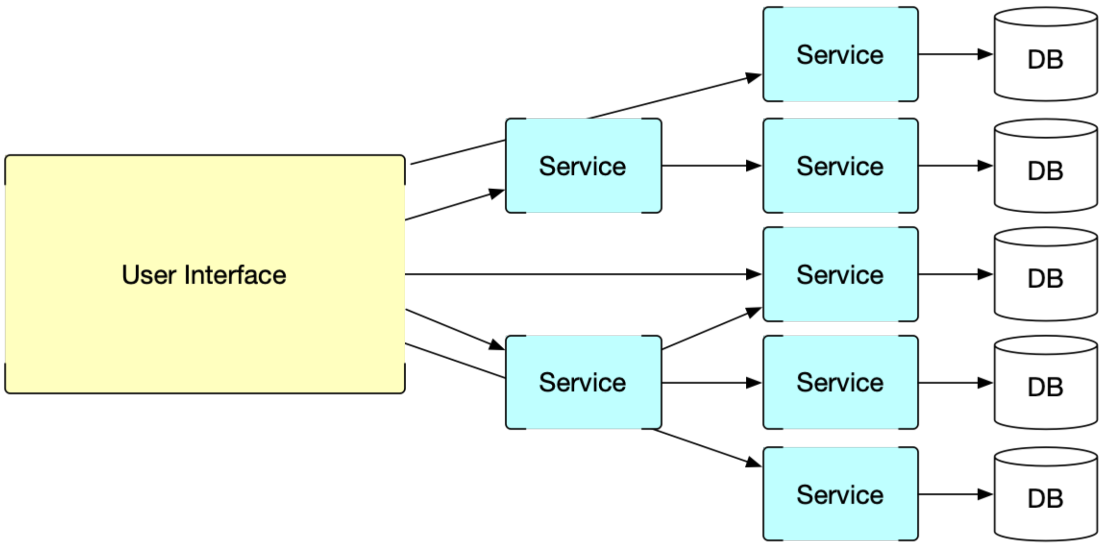

Figure 2.2: A microservice application

2.1.4 Virtual Machine

A Virtual Machine (VM) is an older concept than the concept of containers [15]. Virtual Machine (VM)s virtualise the whole Operating System so it is possible to run different systems on the same machine [16]. This enables administrators to provision machines to their needs. Virtual machines work with a hypervisor. This hypervisor can be directly installed onto a machine or installed within a host Operating System (OS). This hypervisor acts as a resource provisioning tool and creates interfaces for the OS to interact with the hardware. Onto the hypervisor various virtual machines with different OS are deployed, which are completely isolated to each other. Host OS virtualisation often is realised on client machines, while baremetal virtualisation is used on servers. Figure 2.3 shows a visualisation of the differences between the virtualisation methods. On the left side, the host virtualisation, mostly used on client machines is shown. A host OS is installed on the computer. This host OS runs a hypervisor as a service which hosts multiple guest OS. On the right side, the baremetal virtualisation, the hypervisor is directly installed onto the machine hosting multiple guest OS. The baremetal virtualisation uses one layer less then the hostOS virtualisation so it is more efficient but there is a must to use a VM to use an OS. A hypervisor can assign various hardware parts to solely be used by distinctive VMs; e.g.: a specific Central Processing Unit (CPU) core or a Network Interface Controller (NIC) is reserved for one specific VM and not used by another one.

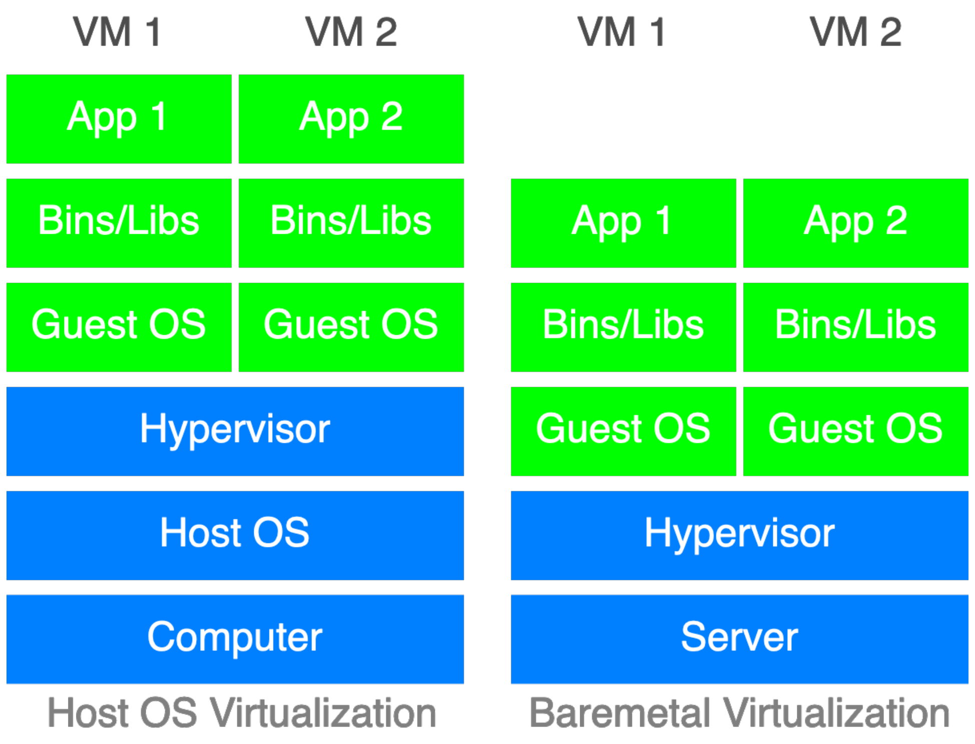

Figure 2.3: Host OS virtualisation vs. Baremetal virtualisation

2.1.5 Container

A container is a way of virtualising a host but it is different to a VM. A VM runs on top of a hypervisor and this hypervisor can host multiple VMs with different OS. A container is installed on an OS. A VM virtualises and isolates everything, a container only virtualises the user space [17] and shares the kernel space with other containers. Containers do not require any hypervisor engine to run, but use the system calls of the distinctive OS. This behaviour saves resources (CPU load, memory, disk space, etc.) but increases the attack vector of a virtualised host [18]. Also, container technology limits the usage of various OS as it uses shared system calls. The most common implementations work on a Linux system and are able to run different distributions in container (e.g.: alpine, debian, centos, etc.) but does not allow to run a system depending on a different kernel (e.g.: BSD, DOS, etc.). Figure 2.4 shows a container based virtualisation. A host OS is placed directly onto a server. The host OS runs a container engine which runs various container images with various apps. Every container shares the kernel space and only has the user space isolated. Also the different containers on a machine share the same hardware resources. For example, this means that every container has access to all NIC on the same machine. In comparison monolithic software runs every application in the same user space and security lies in the correct programming of the software and correct memory and cpu handling.

Container target the use of multiple system distributions on the same kernel. Microservice make use of the concept of resource sharing and isolating, and therefore often base on container images. If a microservice system needs multiple various kernels, one VM or physical machine is needed per kernel.

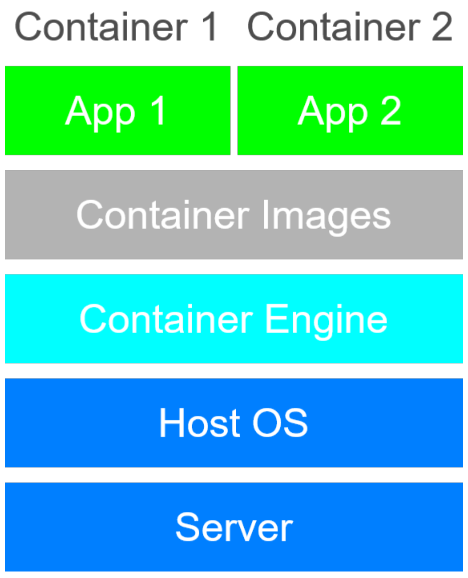

Figure 2.4: Schematic container architecture

2.2 Property Evaluation of Microservices

“The essence of microservices is that they are (or compose to form) highly modular, distributed systems, reusable through a network-exposed API. This implies that microservices inherit advantages and disadvantages of both distributed systems and web services” [2]. To distinguish microservices from other software architecture, four characteristics can be distilled:

- Distributed: Microservices build on other microservices and communicate a lot over a network
- Network: Microservices communicate over a network with other microservices
- Modular: Microservices should offer a fine-grained API that allows to use it for various applications
- Encapsulated: Microservices are isolated from other services; they only exchange information over predefined interfaces, but do not relay on a specific infrastructure. Microservices working together could be written in a complete different language.

Figure 2.5 shows the differences in the structure of monolithic applications and applications with microservices. The user interface, the logic, and the data access is contained in the same package. While monolithic services contain every part of the system in it, microservices are seen as multiple independent programs designed to work together. Every logic function and data access is split up into an own microservice (e.g. in a container). This makes it highly scalable and also very chatty.

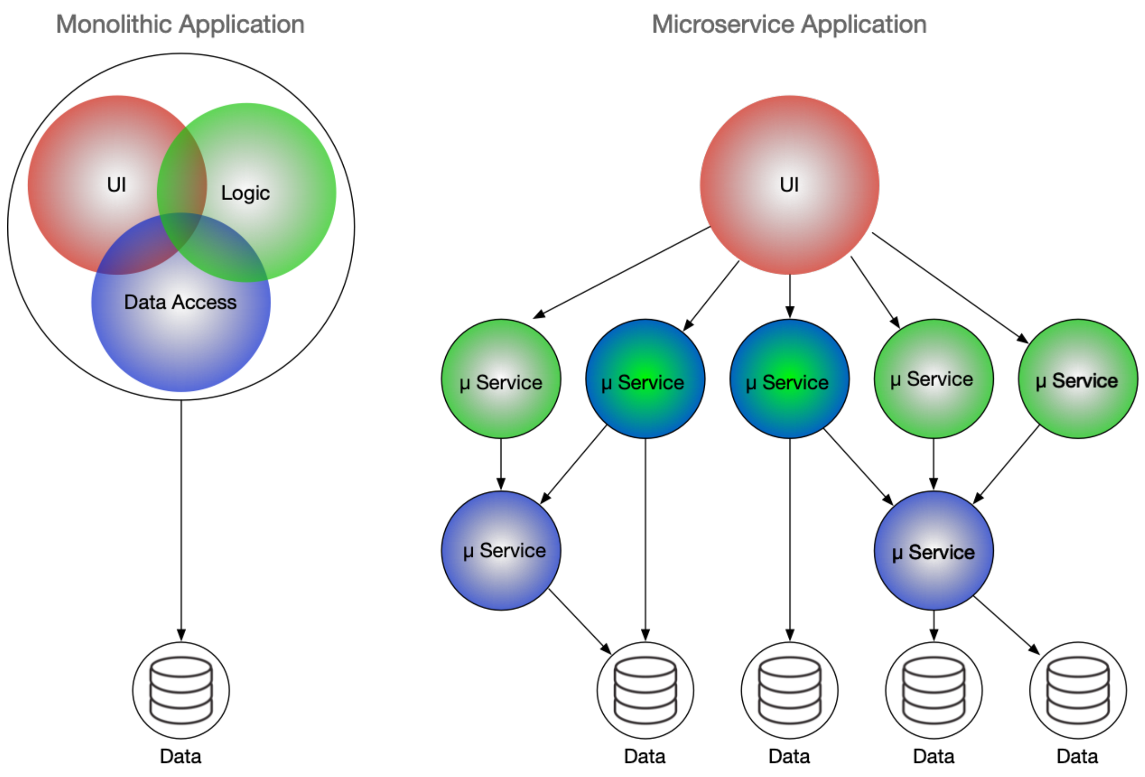

Figure 2.5: Monolithic vs. Microservice architecture

2.2.1 Advantages

As in every property evaluation there are advantages and disadvantages of specific objects. To start with the good news, the advantages will be examined.

Maintainability

As every microservice is independent from another service and communicates with defined interfaces a developer of a specific service does not need to know the insights of another service. The only needed information is the given input and needed output of a service. This enables the developers to either quickly patch a service or to replace it with a completely different one. This is very useful to keep an application state-of-the-art and helps to get rid of unsupported carrier OS. These small applications lead to a faster and more frequent software release [19].

Heterogeneity

The same reason as the ones written in the section on maintainability gives the freedom to write each microservice in a different programming languages. “Developers can freely choose the optimal resources (languages, frameworks, etc.) for the implementation of each microservice” [5]. This is useful to exploit the respective advantages of various programming languages. Also, each unit can run on another OS and with other libraries [14].

Development time

Microservices are small services, which can be written in any language wanted and independently from other services. That makes them easy and fast to develop, which leads to a low development time and enables fast application of changes.

Scalability

There are two major forms of scalability in terms of computing power and performance; *scale up* and *scale out*. The respective opposites of these terms are *scale down* and *scale in*. Michael *et al.* describes these forms as followed [20]:

- Scale-up: The deployment of applications on large shared-memory servers (large SMPs).
- Scale-out: The deployment of applications on multiple small interconnected servers (clusters).

To form this in easier words, scaling up means to use a better and faster server while scaling out means to use multiple servers. To boost performance on a monolithic application, scaling up is the only thing possible to do when it comes to hardware, which makes this kind of application very dependent on hardware improvement.
As a microservice application is the cooperation of multiple small services, they can get distributed to multiple machines. Adding more machines to a cluster is defined as scaling out. Scaling out enables the use of cheap hardware and stacking it. Also, adding and removing resources (machines) is much easier when scaling out than scaling up, because often low or no maintenance work is necessary. Scaling out often works without the need of a maintenance window and shutting down services. Microservices take advantage of this architecture [21] [8] [22] as time- and resource- consuming services can easily get multiplied, there is no need to multiply the whole application. Multiple cloudsystems support automatic scaling out/in [23]. This enables microservices to automatically react to high load and distribute relevant services to new allocated machines to improve performance or remove machines with low load and save costs.

Availability

Microservices are designed to deploy one service multiple times in a cluster which makes microservices highly available [22]. Well designed microservices are able to accept workload assigned to non-functional services (service outage, corruption, etc.).

Code Comprehension

As microservices are a smaller piece of software without internal code references to other projects, the comprehension of the code gets much better. As a result security issues are more likely to be seen[3].

2.2.2 Disadvantages

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> Leslie Lamport, 1987

As computer science is often a trade-off between things, most advantages are followed by disadvantages. This section describes some of the distances of microservices.

Asset Management

Microservices result in multiple machines and/or containers. This results in a difficult asset management due to automatic naming and a constant change in network topology [2]. Automated outscaling amplifies this management workload. To ensure a correct working asset management and a resulting visibility of the respective use case of the machines, their working times and the costs when working with microservices, an automated approach is highly recommended.

Data Consistency and Replication

As microservices work distributed over the network, often running multiple instances, keeping track of data changes is a challenge [2]. A problem when working with microservices is to keep data consistent and ensure a correct data replication to all authorized units. This relates to the Atomicity, Consistency, Isolation, Durability (ACID) properties. As there is no shared database, but all data is owned by the services themselves, ACID is difficult to achieve in distributed systems [24]. Tanenbaum *et al.* proposes an approach

[3]https://docs.microsoft.com/en-us/archive/msdn-magazine/2007/november/code-reviews-find-and-fix-vulnerabilities-before-your-app-ships

called two-step-commit. This allows several changes to a system as an atomic operation and also allows a rollback if a failure occurs. Eventual consistency is a weaker form of consistency. It works via updating data in regular time frames of various services guaranteeing a maximum age of data.

Isolation

Microservices work with their local data, only pulling input and pushing results to the network. The service itself works independently from other services and therefore does not check for system health. “The overall system state is unknown to individual nodes” [2] [24]. This applies to all system problems, including corrupted and hacked systems; individual units cannot determine if the system has been corrupted.

Operational Overhead

Microservices create a huge operational overhead compared to monolithic services. Ueda *et al.* reports an up to 80% lower performance when comparing a microservice application to a monolithic application running on the same hardware [25].

Also, this overhead in infrastructure leads to a lot of time investment in infrastructure before and during development. An extra team is needed, solely to support developers in infrastructure requirements. Microservices require an automated approach and therefore service discovery and management infrastructure components [26].

Architecture Overhead

Microservices create a need to make multiple design choices [2] and creates a huge architecture overhead. Microservices need an extra management infrastructure to run securely and reliable functional services.

Fault Tolerance

Fault toleration is the ability to operate although a partial failure has occurred on a system [24]. A failure producing another failure is called a cascading failure. These often lead to overall system failures making it unproductive. To prevent this, a circuit breaker can be implemented [27] [26], which will for example throttle down connection attempts to a broken service, or enforce connection timeouts. It is not trivial to establish fault tolerance in distributed system.

Documentation

Documentation of microservices itself is easy to accomplish, as only the developer working on the microservice is needed to create a documentation of this service. To document the whole system, which service is in charge of what, which data is needed on which interface and which data should get returned is a major challenge. A sudden change in the architecture of a service can impact the whole system.

Software Testing

Software testing and fuzzing gets complicated with distributed systems [26]. To avoid downtime due to cascading failures (see *Fault tolerance*) automated, regularly, broad testing of software components, to detect potential system crashes early, is necessary. To test resilience of systems in productive environments a new approach called *chaos engineering* is getting popular [28]. Chaos engineering automatically injects faults into the system (e.g. randomly killing a unit) to encourage developers to build resilient services which are able to self heal and resist partial failures.

Privacy constraints

“Since each microservice typically accesses different data, while composing complex applications it is hard to monitor which data are getting accessed in the entire application workflow” [29]. As microservices are a distributed, data intensive architectural style, it is very hard to ensure privacy guidelines. This means microservice architecture on its own, the multiple connected services sharing data, as well as the common implementation of microservices in cloud environments.

2.3 Trends & Challenges of Microservices

As in every technology branch, automation is the key desire to push new technologies [26]. Microservices enable developers to automate big parts of their infrastructure and therefore push the adoption of microservices. A further trend is serverless computing. Functions are written but not assigned to any machine in the first place. It is chosen, depending on whether it is time critical or not, and the computing of the functions is billed via computational time. This is only possible with micro and nanoservices.
Although microservices are already established, there is still no standard or best practice on how to build microservices [26]. Therefore, there is also no guideline on how to transform a monolithic service into a microservice. There is also no best practice on how much granularity microservices and its APIs should have and no guideline on how to communicate with other services. This lack of standardisation leaves a lot of

room for developers to create services and therefore to a lot of different services. Also, as a result of this lack, many security vulnerabilities which were though to be extinct in serious services reappear, as developers mostly have not been concerned with architectural security challenges in the past. For example, to store data in a microservice architecture is not a trivial thing, and not standardised. Storing data in a traditional, centralised way does not comply with microservice requirements and can lead to serious problems, including data inconsistency. Although proposed approaches already exist[4], there is no approved guideline yet.

2.4 Core Concepts of Security

To understand microservice security, some core concepts of network and IT security should be understood. The following sections briefly explain chosen topics, which are essential for the following chapter of microservice security when it comes to data preservation and tampering protection as well as identifying, authenticating, and authorising various users and services.

2.4.1 CIA Principle

The CIA principle[5] refers to the terms confidentiality, integrity and availability of information and has nothing to do with a three-letter-agency.

Confidentiality refers to the state that specific information is only visible to specific people. Confidentiality can be ensured via encryption of information and share the key with entitled persons or services, or via assigning read and write rights to these individuals. Authentication is part of confidentiality, as there is no confidentiality without authentication.

Integrity means that information is not manipulated without anyone noticing it. Integrity ensures that files and data remain in a state that is expected by users and is protected against unauthorized tampering through users or services. Hashes of files can ensure the alerting of tampering of information.

Availability is the concept of providing the right data when necessary.

All three terms complete each other and should be balanced [30]. The CIA triad (Figure 2.7) represents it. Market ready solutions for confidentiality often include integrity protection, because confidentiality as well as integrity protection mostly is based on cryptography. These aspects are often covered identically well. Availability needs some cryptography aspects but is mostly covered through a replication concept. As confidentiality and integrity, and availability need two different approaches to cover the aspects, availability

[4]`https://auth0.com/blog/introduction-to-microservices-part-4-dependencies/`
[5]`https://www.forcepoint.com/cyber-edu/cia-triad`

is often either forgotten, or too much weighted at the expense of confidentiality and integrity and therefore often not balanced in the triad.

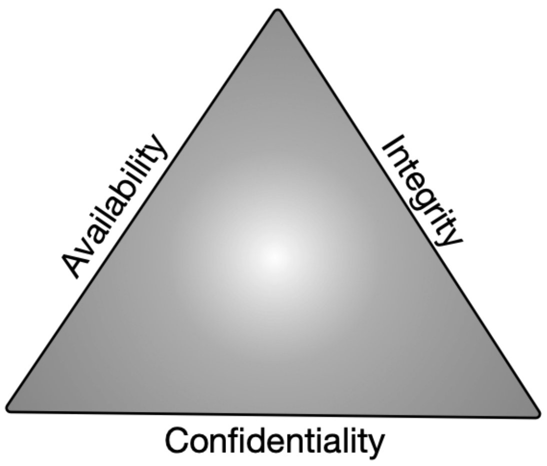

Figure 2.6: CIA triad - All three terms must be balanced in the architecture and security measures to guarantee a good working information security.

2.4.2 Identification, Authentication, Authorisation

The words identification, authentication, and authorisation are widely used in IT security[6]. Yet, only few people can tell the difference between all three of them, and correctly assign these principles to projects and software.

Identification is the claiming of an identity. This can be compared to saying a name, or typing in an email address into a login form. Identification alone does not prove anything and cannot be used to grant privileges and permissions. Identification is solely a intermediate step, which is used to fail-proof authentication and increase precision of such algorithms. Authentication algorithms can work without the identification step as well, which some of them do.

Authentication is the prove of an identity. This happens by showing an ID or typing in a password for example. The process of authentication can be done via three factors: something you know, something you have, something you are. Knowing can be handled through a password, having via a security token (USB drive, smart-card, etc.), or via a biometric scan (fingerprint, retina scan, etc.). Multifactor authentication works when combining at least two of the above described ways (e.g.: Smart card with PIN, Smart card with fingerprint). Authentication is necessary to ensure confidentiality. Authentication works with and without

[6]`https://www.kaspersky.com/blog/identification-authentication-authorization-difference/37143/`

identification. When someone attempts to authenticate without identifying, the proof of authentication is checked against the total of all users or services which results in a 1 to N check ($O(N)$). This increases computing time and error rate, which decreases security. Identifying before authenticating results in a 1 to 1 check ($O(1)$). This decreases computing time and error rate, which results in increased security.

Authorisation is the process of assigning privileges to entities (e.g.: read permissions on specific files, access permissions of various programs). Several approaches to assign permissions exist, including Access Control List (ACL), Role Based Access Control (RBAC), Attribute Based Access Control (ABAC). ACL assigns permissions based on identities. In the RBAC approach, permissions are assigned to roles, which are assigned to identities. ABAC is an extension of RBAC as it assign permissions to attributes, which are assigned to roles and identities. The ACL approach is the easiest approach, but also quite coarse, while ABAC is the most complex approach but also the most fine grained.

2.4.3 Public Key Cryptography

Public key cryptography, also asymmetric cryptography, works with two keys, a private and a public key [31]. The names of the keys already describe where they are stored. When encrypting or signing something, this can be done with both keys. Always the opposite key is needed to decrypt or check the signature. Therefore when encrypting something with the public key, only the private key can decrypt it; when signing something with the private key, only the public key can check it. Encrypting some thing with the public key ensures confidentiality, as only the bearer of the private key can decrypt it. Signing with the private key ensures the authenticity of data, as everybody can check that this has been signed with one specific private key.

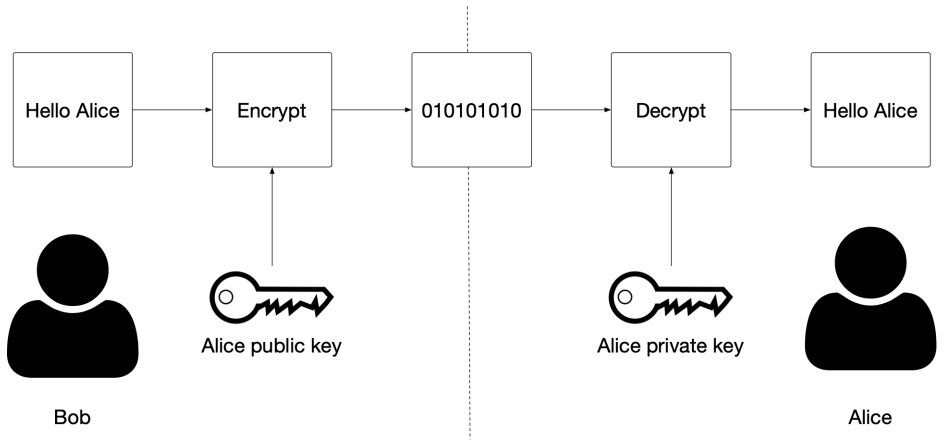

Figure 2.7: Public Key Cryptography - Bob uses Alice public key to encrypt a message, which can only be decrypted by Alice with her private key.

Public key cryptography has been the key concept of a modern secure IT and web. Symmetric cryptography is much faster than asymmetric cryptography, but asymmetric cryptography solves the problem of key exchange [26]. Security of the transport layer of the Open Systems Interconnection (OSI) system is entirely based on asymmetric cryptography in Transport Layer Security (TLS), as well as it is with already deprecated, but unfortunately still in use, Secure Socket Layer (SSL). TLS provides confidentiality and integrity protection. Previous TLS versions allowed a variety of insecure and obsolete crypto modes which were removed in the actual standard TLS 1.3 [32]. Therefore an adoption of the new standard is recommended.

Asymmetric Cryptography works best with an infrastructure, a Public Key Infrastructure (PKI). A PKI issues digital certificates, which bears both keys, or just the public one. They are used to execute all asymmetric cryptographic operations. The PKI is used to validate certain certificates, as they establish a trust chain [33] [34]. The validity of a cryptographic operation is reviewed by its technical correctness and if a trusted, valid certificate was used. A core problem of this infrastructure is the fast and correct distribution of the Certificate Revocation List (CRL). As running a productive PKI is non trivial, short comings of this approach need to be understood [35].

2.4.4 Authentication and Authorisation in Distributed Systems

R2D2, you know better than to trust a strange computer.

C3PO, 1980

To ensure authenticity and authorisation in modern web and IT, delegated approaches have been developed. These approaches have been developed to increase user convenience, as it enables them to use a single account for multiple services. Prominent examples are OpenID, OAuth, and OpenID Connect (OIDC). As OpenID itself has been deprecated, it still plays an key role in the OIDC protocol[7]. These protocols also introduce interesting applications in a distributed microservice approach for inter-service communication [26].

OAuth 1.0

OAuth is an authorization protocol, not intended to use as an authentication protocol [36]. Therefore OAuth assumes that requests have been already authenticated.

OAuth 1.0 works with two credentials within each client request: for the resource owner and for the client itself [37]. Before accessing a resource, a client needs to request for permission by the resource owner via the authorisation server [26]. If successful, the server issues an access token to the requester. This token

[7]`https://openid.net/developers/libraries/obsolete/`

represents the approval of the resource owner and enables the client to request resources from a Service Provider (SP). The client credentials enable the participating parties to authenticate a client and ensures message integrity.

OAuth 2.0

OAuth 2.0 only utilises one credential, the resource owners [38], there is no client authentication anymore. OAuth 2.0 is simpler than OAuth 1.0, but depends entirely on TLS to guarantee authenticity and integrity [26]. Both versions, 1.0 and 2.0, depend on TLS to guarantee confidentiality. Also, as there is no client authentication, OAuth 2.0 now less than ever should be used as an authentication protocol. Also, as it introduced more optional arguments, it led to interoperability issues between multiple providers. OAuth 2.0 introduces four use cases, but only two are popularly used, whereas one is often used falsely: implicit flow and authorisation code flow.

The *Implicit Flow* is intended for third parties, that are not able to handle confidential data securely [26]. Figure 2.8 visualises the implicit flow. The user sends his credentials and the ones of the requesting client to an authorisation server and receives an access token. Further it sends the access token to the resource server and receives the requested resources [38]. The implicit flow is much easier than the authorisation code flow, but not as secure as the latter, as it exposes internal access token to untrusted parties, as for example the user.

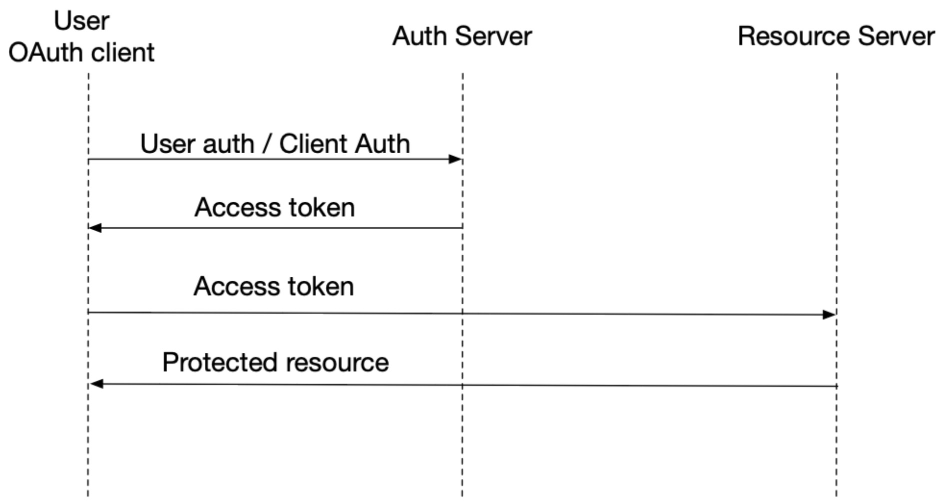

Figure 2.8: Implicit flow of the OAuth protocol [38]

The *Authorisation Code Flow* targets trusted clients. The authorisation code flow is more secure, as it never exposes an access token to any non-trusted client or user. The only party getting an access token is a web application (e.g.: API gateway) running on a trusted web server. Figure 2.9 visualises this flow. The user sends his authorisation request with the authorisation request of the used client to the authorisation server. The server returns the authorisation code for the client to the user. The user sends this code to the client. The client uses this code to request an access token for the user and receives it. The user sends a request to the client, the client combines it with the access token and sends it to the resource server, which return the resource back to the client and then to the user. The user never gets any access token.

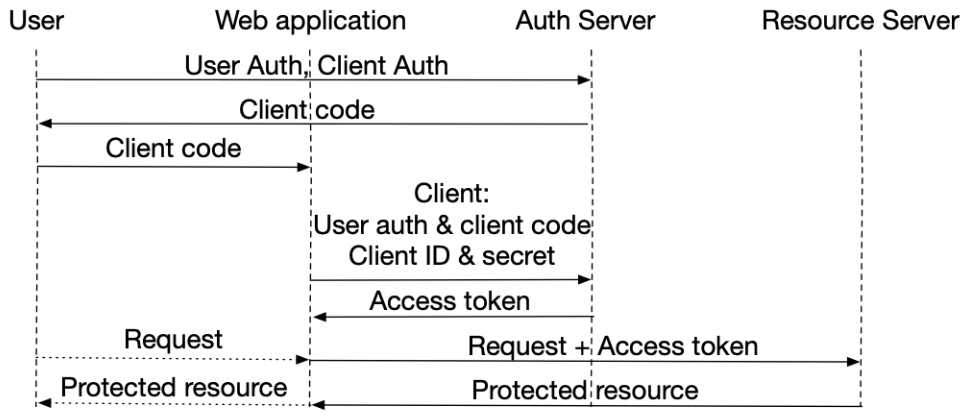

Figure 2.9: Authorisation code flow of the OAuth protocol [38].

OpenID Connect

OpenID Connect (OIDC) is a authentication and authorisation protocol, combining OpenID and OAuth 2.0[8]. It is well used as a Single Sign On (SSO) protocol. OIDC is widely used as so called "social login" ("Login with Facebook", "Login with Twitter", etc.). OIDC includes Identity Provider (IdP), which are working as authentication and authorisation server, and SP also called Relying Party (RP) to provide requested resources. The protocol offers some extended features to improve security [39], which might decrease privacy:

- An RP can request further information about a requesting user, such as email address, phone number, login details, etc. from the IdP or included in the access token.
- Tokens of the IdP and the RP must be signed and encrypted.
- RP must authenticate to IdP via digital signature, or Hash-based Message Authentication Code (HMAC) and shared secret.

The request flow is very much similar to OAuth 2.0.

[8]https://openid.net/connect/

3 Related Work

After clarifying the basics of this topic, an introduction to the state of the art is necessary. This chapter shows related work and what has already been done.

Yarygina *et al.* created a first survey about microservice security [2]. They gave a broad overview of security thoughts and their implementations. Their focus lies on authentication and authorization of users as well as services. They outlined a model which follows the OSI model and summarised some core concepts. Yarygina *et al.* also created a microservice based banking system as a proof of concept and tested performance impacts of various common security implementations in microservices. Their experiments included the usage of a self-hosted PKI to enable Mutual Transport Layer Security (MTLS) connections and a reverse Security Token Service (STS) to issue security tokens.

Dragoni *et al.* gave a historical overview of microservice evolution and gave an prospect to what microservices will look like in the future. They addressed key aspects in the development that have been solved and will be solved through microservices [5].

Almeida *et al.* did a basic survey about microservice security and privacy in a cloud context in 2017 [13]. The paper gives an overview about elements to consider for developing microservices and describes the architecture of those.

Otterstad *et al.* evaluated microservice security approaches on low-level attacks [14]. They suggest a wide heterogeneity of used systems in every way. They suggest the usage of various OS, and various exposure levels and corresponding access profiles.

Dragoni *et al.* gave a broad overview about microservice scalability and why this is a key property to software development [22].

Tanenbaum *et al.* wrote a fundamental guide of distributed systems, describing the principles and paradigms, which can be applied on microservices as well [24].

Yarygina also did a comprehensive doctoral thesis about microservice security [26]. She concluded that there are differences in a SOA and microservice architecure, as well as automation is the key feature that needs to be considered in every step in microservice development.

Vistbakka *et al.* analysed microservice architecture in terms of privacy and made a first definition of privacy-preserving microservices [29] [40].
Fetzer introduces an architecture for critical applications in a microservice approach [41]. He, as well as some others, proposes to take advantage of hardware related security features like Software Guard Extensions (SGX) to protect microservices.
Casalicchio *et al.* created a comprehensive survey about Continuous Integration (CI) security and usage [42]. In combination with Ullah *et al.*'s work about Continuous Deployment (CD) security [43], it is a great overview about DevOps protection.
Pahl *et al.* created a communication monitor to detect anomalies in the communication of microservices [44]. They created a service between every communication, basically working as a Man-in-the-middle (MITM) getting trained and then blocking all non-expected communication.
Sun *et al.* introduce a new API flow for the network hypervisor in cloud based microservice systems to build a network monitoring and policy enforcement [45].
Hahn *et al.* surveyed the security of common service mesh implementation and tools [46]. They basically concluded, that service mesh tools are placed in microservice environments and behave like microservices, therefore basic microservice security needs to be applied to service meshes as well.

4 Security Challenges regarding Microservices

Monolithic services focused security on single points [47]. “Monolithic services have a clear boundary and encapsulate their communication” [13], therefore security vulnerabilities are obscured to the inner layers of the system [48] [49]. Microservices rise different security challenges to solve than traditional services, e.g.: communicating sensitive data over the network, whereas monolithic services used the Random Access Memory (RAM) and security approaches of the RAM for it. Thus microservices depend on multiple units communicating with each other, trust establishment between them is an essential challenge to be solved. As microservices are a fast growing and fast shrinking application approach, infrastructure and all services monitoring and securing infrastructure need to be highly automated. Also, the fast change in software development from monolithic services to microservices and their Web-API behaviour leads to published code, that was never meant to be exposed [2].

4.1 Layers of Security Challenges

To examine the security challenges of microservices Yarygina *et al.* propose to address them based by categories: hardware, virtualisation, cloud, communication, service, and orchestration. These categories will be explained and their challenges investigated. These categories can be viewed like the OSI layers. Vulnerabilities threatening one specific layer will likely compromise all other layers below as well as seen in Table 4.1 [2]. The principle of system security layers will be introduced. When having control of one layer, one has also control over all layers below the controlled layer and absolutely no control over the layers above; e.g.: One has control over the communication layer, therefore one has also full control over the service and orchestration layer, but absolutely no control over the cloud, virtualisation and hardware layer.

4.1.1 Hardware

All software security depends on strong, safe hardware. Bugs like Meltdown [50], Spectre [51] and ZombieLoad [52] enable attackers to create powerful side-channel attacks. Also weak Random Number Generator (RNG), tampered Trusted Platform Module (TPM) and poorly designed CPU [53] compromise the security of the whole system. Weak RNG lead to weak random numbers. These random numbers are essential when using cryptography (e.g.: creating keys) and therefore rely on a good entropy. TPM are cryptographic chips, especially designed to create and save cryptographic keys and operate with them on hardware level (e.g.: full disk encryption). Backdoors in these chips or in other parts[1] of the hardware placed by manufacturers or intelligence services can be introduced at the point of manufacture[2] as well as later on[3]. There are only a few things to mitigate these threats[2]:

- Designing own hardware
- Diversification of hardware
- Use of Hardware Security Module (HSM)

Designing own hardware, especially security chips is an expensive and non-trivial thing. This only pays off for big companies in need of a lot of hardware (e.g.: cloud provider). A diversification of hardware [54], especially varying hardware producers, is a powerful method to mitigate specific hardware vulnerabilities. This does not help against fundamental design issues as in Meltdown or Spectre, but helps against specific backdoors from manufacturers. HSM, e.g. smart cards or security tokens, are a profound way to store cryptographic keys outside of the usual system.

4.1.2 Virtualisation

There are many ways to virtualise services. From sharing resources and isolating services (VM, container), separating processes (OS memory management) to testing applications and files (sandboxing) many reasons exist to justify virtualisation. For all causes virtualisation is fundamentally done by isolating software processes and sharing hardware resources. Threats against virtualisation are summarized to attempt to break virtualisation, therefore to get access to isolated services [55], which is called in general "Virtual machine escape". "Attacks include: Sandbox escape, hypervisor compromise, and shared memory attacks; also, use of malicious and/or vulnerable images is another serious concern" [2].

[1] `https://www.slideshare.net/endrazine/defcon-hardware-backdooring-is-practical`

[2] `https://www.extremetech.com/computing/133773-rakshasa-the-hardware-backdoor-that-china-could-embed-in-every-computer`

[3] `https://www.infoworld.com/article/2608141/snowden--the-nsa-planted-backdoors-in-cisco-products.html`

To prevent and mitigate these imperilments, Yarygina *et al.* propose to prefer stronger isolation virtualisation, to not share library and hardware caches, to verify the origin and integrity of images and to obey the principle of least privilege [2]. Pearce *et al.* further recommends to regularly patch and update the hypervisor and execute integrity checks [56]. Virtualisation can also be a threat when not using any virtualisation. Malware can place a hypervisor as another layer between any layer of code recording and intercepting any request. One of the first proof of concepts was blue pill[4][5]. Also a reverse sandbox can help identify threats and mitigate them [2].

4.1.3 Cloud

Cloud computing opens a new chapter of security hopes and nightmares. Cloud services serve as a remote infrastructure which can be bought at various service levels. This flexibility is highly attractive for administrators and accountants, at least at the first glance. Each level guarantees different services and enables various access points to administrators and attackers. Infrastructure as a Service (IaaS) provides access to the machine layer, meaning an administrator can install and configure his (virtual) machine as wanted. Platform as a Service (PaaS) enables an administrator to use a platform (e.g.: a webserver) and configure this webserver as wanted. Software as a Service (SaaS) provides access to software (e.g.: Office 365) where no client software has to be installed, as everything runs in the cloud. Cloud security has to be seen as a completly new field in information security as this is a highly complex topic [57]. As for example, when correctly configured, cloud services can guarantee to always be up-to-date (PaaS, SaaS), these thoughts can lead to misconfiguration when buying other levels of service (IaaS)[6].
To address cloud security issues, one can follow the CIA principle:

- In terms of *C*onfidentiality, the cloud solves nothing out of the box. When using a cloud to store and process data all of the already known best-practices and pitfalls have to be considered. Additionally the cloud creates another threat: the cloud provider. A cloud provider has unlimited access to every system in its infrastructure. This includes all data. When applying confidentiality policies traditional actions like access management have to be combined with strong encryption methods, where the keys are not stored inside the cloud. Also, the communication from the workplace to the data/cloud should be regarded. A Virtual Private Network (VPN) can help to increase chances to ensure confidentiality

[4]`http://web.cecs.pdx.edu/~wuchang/courses/cs492/BluePill.pdf`
[5]`http://conference.hackinthebox.nl/hitbsecconf2006kl/materials/DAY%202%20-%20Joanna%20Rutkowska%20-%20Subverting%20Vista%20Kernel.pdf`
[6]`https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-update-azure-service`

and integrity[7].

- "*I*ntegrity provides assurances that data has not changed"[8]. With a cloud provider having access to the infrastructure and data this is an attack vector which strongly depends on trust, that this one is not being violated. Also a point to be considered is the distributed data set, which is not under control of a local administrator. Systems must be resistant enough to survive sudden update and transfer procedures and must ensure that data is not getting corrupted. As the cloud providers update and transfer data at their decision, there is no way to time infrastructure changes, system changes and maintenance windows [58].
- *A*vailability is the last key property in the CIA triangle. All cloud services advertise their high availability and backup solutions in case something is being attacked, compromised or crashed. Another point of availability is the fact, that companies, when using the cloud, do not have the sovereignty of their data anymore, and in fact lost the availability. When it comes to gaining back data and retrieving them from the cloud, they rely on agreements with the cloud provider and politics of the involved countries. This is highly relevant for critical infrastructure and for companies with a high-value-product. A quote of Frank Rieger regarding the cloud is: "Your data is somewhere else, and you do not know where"[9].

Another factor is the rising privacy issue which will not be discussed in this paper. Although most work only focuses on cloud security, Abdullah *et al.* created an access control concept which aims to protect the privacy of the user while preserving the CIA-triad on the data [59].

4.1.4 Communication

Due to their architecture, microservices need to communicate a lot with other services. This communication mostly happens over the network stack [5], which enables a great portability, therefore microservices are not bound to any cloud or hardware provider [42]. As it is common in the DevOps-era and brings a lot of advantages but also disadvantages, microservices use API to communicate with other microservices [27], including Representational State Transfer (REST) APIs and Simple Object Access Protocol (SOAP) [60]. Therefore the next layer of attack vectors are based on the communication of microservices [2]. "The microservices must assume that any input encountered is hostile" [14]. Microservices are communication over an insecure network to users and other microservices and must be aware that users may input dangerous

[7]`https://clearchoiceinc.com/ensuring-confidentiality-in-the-cloud/`
[8]`https://cybersecurityglossary.com/integrity/`
[9]`https://www.faz.net/aktuell/feuilleton/debatten/ueberwachung/snowdens-enthuellungen-sind-ein-erdbeben-12685829.html?printPagedArticle=true`

input and other microservices may be compromised. Attack threats to this layer are quite the same as the typical network problems like MITM, Denial of Service (DoS), and Distributed Denial of Service (DDoS) attacks, including eavesdropping, spoofing, and session hijacking extended with protocol attacks on TLS encryption; e.g. Heartbleed [61] and POODLE [62]. A security threat could be the use of clear-text end-to-end traffic between services.

The approach to communicate over network interfaces causes new attack vectors to arise [13]. This is one of the major challenges in the microservice approach [63] [47]. To mitigate threats the global usage of standardized cryptotools like an up-to-date TLS implementation is recommended; also when only communication in the same cloud environment[10], company network, or on *localhost*. Also, a good security perimeter concept should be in place. A quick win will be the usage of different credentials across the services [2], as well as the usage of microsegmentation as later described.

4.1.5 Service

Microservices suffer from the same kind of application vulnerability as other applications. Fast feature deployment leads to user happiness and security vulnerabilities. OWASP releases the ten most common application vulnerability categories yearly[11]. Despite the fact that OWASP stands for Open Web Application Security Project, this rating represents all computer applications. Some of them are represented most years: injections (SQL, OS, LDAP, etc.), broken authentication and access control, data exposure, Cross-Site-Scripting (XSS), broken deserialisation, insufficient logging, and usage of unpatched and/or outdated libraries.

Static and dynamic code analysis including a manual code review helps in getting rid of most security issues [2]. Also, the correct usage of input validation, error handling and a good documentation of API helps. Additionally to the full disk encryption at the cloud layer, data encryption in databases or in files helps to slow down and repel attackers. Orchestration can be useful to visualise dependencies and keeping them up to date[12][13].

[10]https://www.darkreading.com/cloud/82--of-databases-left-unencrypted-in-public-cloud/d/d-id/1328966

[11]https://owasp.org/www-project-top-ten/

[12]https://medium.com/better-practices/conquering-the-microservices-dependency-hell-at-postman-with-postman-part-2-7c825d576947

[13]https://simplabs.com/blog/2019/04/24/dependency-updates-for-gitlab/

4.1.6 Orchestration

Orchestration is the management of the various service units and nodes[14]. As microservice units need to spawn and disappear automatically, this needs to be orchestrated. Due to a continuous start and stop process, the infrastructure and network changes as well. Attackers could use this to discover services [27], or create central points to reach all services in the network [64]. Common attacks are the registration of a malicious node and the redirection of traffic [2]. Orchestration is a special case in this layer principle. As every orchestration framework obviously depends on the service it is hosted in, the orchestration also influences the services. It is not entitled to create them, but could introduce malicious services to the system and destroy critical ones. [42].
An effective protection of the orchestration platform is necessary but not well researched. A best practice so far is to use common protection methods like a strong password for this services only, encrypted connections and encrypted configuration files.

4.2 Types of Security Measures

Security measures are divided into three types: preventive, detective and corrective. There is a huge difference between them, as when applied preventive measurements correctly, there is theoretically no need for detective and corrective measurements. But to repair an already compromised system, one must detect the maliciousness and correct it. Without detection, correction would be useless and vice-versa.

4.2.1 Preventive Measures

When implementing preventive measures, this is called security by design [26]. Some basic approaches are:

- Minimize attack surface area - Limit number of interfaces, APIs, proposed services to an absolute minimum
- Security by default - As configuration settings are mostly not changed, the default settings should be the most secure ones.
- Defense in depth - No component should be trusted and everything should be validated (see subsection 4.3.5).
- Least privilege - Every entity is privileged with required rights only. A dedicated user with a fine grained rights management for a webserver is more secure, than just executing it with the root user.

[14]`https://www.forbes.com/sites/forbestechcouncil/2019/10/28/orchestration-of-microservices-monoliths-people-and-robots/`

Layer	**Threat examples**	**Mitigation**
Hardware	Hardware bugs affect the whole system and cannot be patched without changing hardware. Hardware bugs undermine every software security solution. Hardware bugs and backdoors can be introduced at manufacturing as well as at shipping time.	Designing and manufacturing of own hardware; diversification of hardware.
Virtualisation	Sandbox escape, hypervisor compromise and shared memory attacks affect the security of a system.	Strong isolation deployment, no shared memory and hardware cache, verifying of images before deployment; principle of least privilege.
Cloud	Unlimited control of cloud provider to your services; Shared environment with other customers	Contracts and certificates with the provider and trust, as there is often no way to control if a provider acts in accordance with the contract.
Communication	Attacks on network stack and encryption methods; sniffing, spoofing, hijacking, DoS, MITM, Heartblead & POODLE.	Use updated standard libraries, follow security guidelines and best practices, deactivate outdated protocols, segmentation.
Service	OWASP Top Ten; typical web vulnerabilities including injection of all kinds, insecure authentication, insecure deserialisation, and misconfiguration	Code analysis and review, input validation, error handling, good documentation especially for APIs.
Orchestration	Automation of services and coordination attacks, compromising service discovery, inserting malicious nodes.	no words of wisdom yet

Table 4.1: Principle of security layers - A summary of the security layers, their threats and mitigations of Yarygina *et al.* If one layer is compromised, the attacker has control over all layers below as well [2].

Most of the proposed concepts in this work represent preventive measures. As preventive measures prevent damage, most to the attention on security measures should lie on preventive ones.

4.2.2 Detective Measures

Detective measures are needed when an incident has already occurred. Unfortunately, detection is needed to notice an intrusion, but without detection there is no correction. Classic detection products are Intrusion Detection System (IDS) and Intrusion Prevention System (IPS) systems. IDS only detect incidents and IPS also try to prevent them. Another way to detect incidents is the usage of honeypots [65]. They detect malicious access to systems and gain knowledge about these attacks and slow down an attacker. The time between detection and correction is important. As long as an attacker persists on a system, the more damage happens to it. Implementing and using a Security Information and Event Management (SIEM) or renting a Cyber Defence Center (CDC) is an effective way to detect any anomalies and attacks on a system.

4.2.3 Corrective Measures

Corrective measures try to rollback malicious activities and prevent them from happening again. An example could be an updated version of a software [26] or the killing of a compromised unit and the redeployment of a refactored one. Preventing malicious activities from happening again usually means improving security measures, as seen in the example before. Some players however, especially intelligence agencies, are actively attacking attackers in the hope of destroying their infrastructure and therefore preventing attacks from happening again[15]. This nearly undocumented behaviour could lead to a series of collateral damages in critical infrastructure and impacting non-participants, and this raises serious ethical questions. It is therefore generally discouraged from doing so as nearly no consequences of executing revenge-attacks in cyberspace are researched yet.

4.3 Security Thoughts and Recommendations on Microservices

As the different layers of security threats have been discussed, some general thoughts and recommendations on microservice security will be displayed. These approaches can be implemented in multiple ways.

[15]https://edwardsnowden.com/2015/01/16/cybercop/

4.3.1 Containerisation

Containerisation is one of the key features of microservices. Most of the microservices get containerised. It has the great advantage, that services may get packaged in an image [66]. This allows the services to get stopped and started quickly [67]. When a system crashes, its portability allows it to recover quickly, and therefore ensures a high availability [13]. Also the fragmentation of the system in smaller pieces (microservices in container) allows greater availability of the whole system. If one service fails, it does not affect the whole system. When the services are running on different machines or even different locations of different providers, the unavailability of one provider, machine, etc. only affect one part of the system, which can get recovered quickly on another location. Such code portability minimizes downtime [68].

Container although introduces a lot of new security issues, which need to be addressed. The main challenges are summarised to network security, isolation, and data and image encryption [42]. The authors mainly address Docker security, which lacks at image distribution and container control levels. Syed *et al.* claim to have created an ecosystem for containers to provide compliance, privacy, safety, reliability, and governance [69]. The following subsections will explain the major problems in container security. An overview of the challenges and proposed solutions is given in Table 4.2 based on Casalicchio *et al.* [42].

Isolation

Container isolation is based on *cgroups* and *namespaces* in the Unix systems [42]. “Isolation in accessing the resources is fairly guaranteed by the Linux namespace and cgroup, but there is a flaw in how containers share the same network bridge” [42]. In general, Linux OS settings protect the host, but do not protect any container from any other container. Jian *et al.* proposes to inspect the status of namespaces and look for anomalous processes to defend against container escapes [70]. Luo *et al.* conclude that the usage of SELinux and AppArmor is essential for container security. Secure alternatives to Docker are Joinet Triton [72], Charliecloud [73], and Socker [74]. Bastille[16] is an alternative for FreeBSD. SCONE [75], a C-library to encrypt and decrypt files on a file descriptor basis, can also be used to secure containers. SecureStreams [76] is used to deploy and process secure streams at scale [42]; DATS increases isolation at application layer level [77]. In general, Casalicchio *et al.* recommends the usage of SGX enclaves, although they mention that this limits the architecture to the Intel architecture and increases implementation complexity [42].

[16]`https://bastillebsd.org/`

Data encryption

The encryption of image layers is a challenge to protect container data from compromise or a malicious user [42]. To prevent data exfiltration, the usage of Intel SGX enclaves and SCONE[17] is encouraged. Intel SGX are special memory regions which are encrypted by the CPU with special read or write permissions. Many cloud providers provide access to those enclaves. In some implementations it is also possible to define the layers of the filesystem in a granular way which contain sensible data, which requires encryption [78].

Network encryption

Containers enable a great portability, therefore they are not bound to any cloud or hardware provider [42]. Again, the most promising security protection approaches are based on the usage of SGX enclaves. SynAPTIC [79] provides secure and persistent communication based on public keys and the Host Identification Protocol (HIP). The HIP[18] tries to enable a seamless change of devices by a user while access a resource. It does not require any architectural changes and support the usage of Software Defined Network (SDN).

Isolation	Data encryption	Network security
SGX-based isolation	SGX-based volume encryption	SGX-based network channel encryption
Kernel based isolation	SGX-based filesystem encryption	Host Identification Protocol
OS hardening	Image layer encryption at rest and on-the-fly	Channel leakage mitigation techniques
Application level isolation		
Escape attack detection		

Table 4.2: The major challenges of container security and their mitigation proposes [42].

4.3.2 Network isolation

Microservices that run on a shared computing node, naturally also share the network ports of this node. As this shared resource of essential infrastructure in microservice means represents a critical attack surface, some countermeasure must be made. Network traffic encryption as described above is a very efficient way to protect against eavesdropping. Another way is to isolate different networks and network ports from each

[17] `https://scontain.com`

[18] `https://tools.ietf.org/html/rfc5201`

other. This could be done via Virtual Local Area Network (VLAN) in on premise infrastructures or any tool provided by the cloud provider (which will probably depend on VLAN). VLAN is a widely used and trusted mechanism to isolate layer-2 networks [80]. The usage of VLAN and resulting virtual network ports limit containers to only access networks they belong to and makes it possible to assign one network port per container. Isolating networks and port must fit into the automation approach to work and be accepted. SDN are a newer approach which are a flexible and fast solution in defining networks and introducing and isolating hosts [81]. They introduce the configuration of networks to applications via an API. This allows application and orchestration software to react just-in-time to adapting network requirements and is essential when constantly deploying, killing and transferring container on different computing nodes. SDN works via implementing a middle layer API between applications and the networking layer. As seen in Figure 4.1, the applications address the SDN API which addresses manageable switches and virtual switches and creates the desired network environment. Using an orchestration system, which automatically isolates and connects containers based on their description and/or metadata would reduce effort and mistakes of the DevOps administrators. Do not mistake firewalling for network isolation! Firewalling addresses OSI layer from 3 up to 7 [82], network isolation affects OSI layer 2 [80]. Kubernetes for example supports firewalling out of the box, but no network isolation[19]. Libra is a cluster management framework which provides network isolation for security and maximise shared network performance [83].

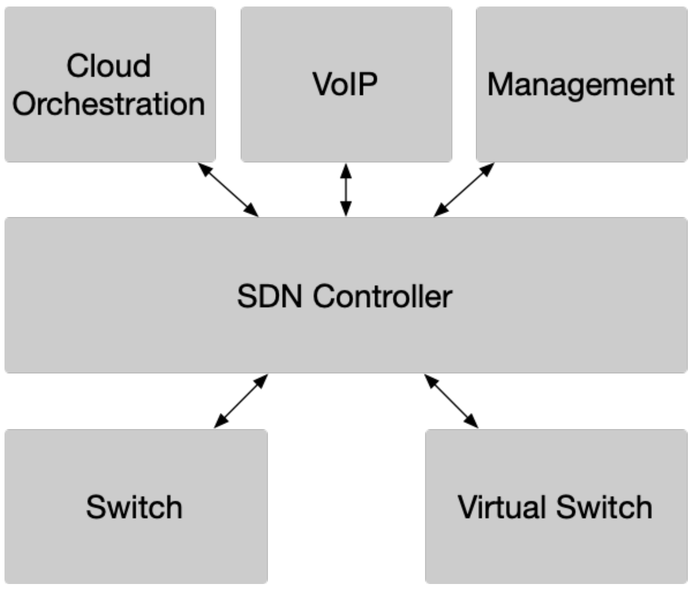

Figure 4.1: Application address the SDN layer for network requirements, which the SDN controller provides via the underlying switches.

[19] `https://kubernetes.io/docs/concepts/services-networking/network-policies/`

4.3.3 Microsegmentation

Microsegmentation differs to SDN and VLAN [84]. While subnets and VLANs need a perimeter to filter requests and connect the nets, microsegmented services can interact and filter request in the same network. Microsegmentation allows to dynamically identify devices and assess its security. It builds device inventories and assigns devices into functional security groups.

Microsegmentation belongs to the zero trust approach[20]. Microsegmentation works via visibility, granular security, and dynamic adaption. It creates different security policies for each namespace. A namespace could be viewed as a dynamic Demilitarised Zone (DMZ). Microsegmentation enforces intra-network security policies which traditional firewalls normally do not control. As seen in Figure 4.2, the firewall is at the edge of the network and only is addressed when working as a gateway to connect different networks. Microsegmentation implementations mostly address the hypervisor, as it has contact to every network packet in a microservice environment. It enables to automate the complete network infrastructure via software. Microsegmentation could be used to isolate microservices containing sensitive data to other ones and therefore meet compliance regulations. Microsegmentation adds a benefit to systems as they result in a smaller attack surface and are able to contain successful attacks in a smaller area. Unfortunately, microsegmentation makes networks and applications more complex which leads to a higher risk of misconfiguration.

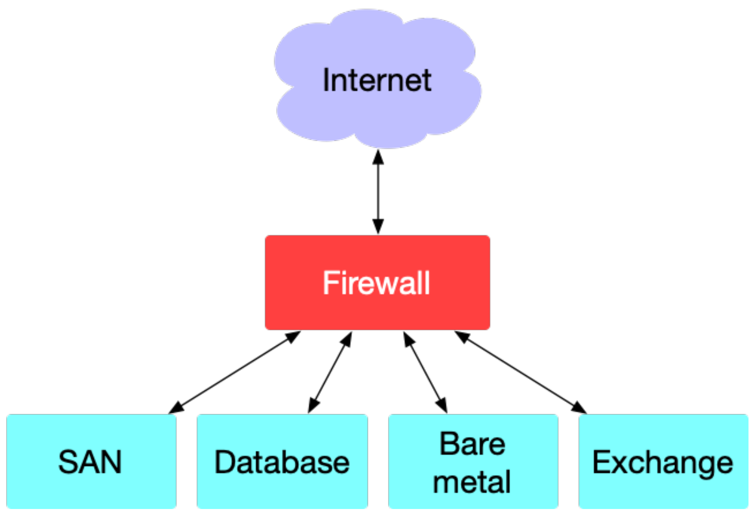

Figure 4.2: Server applications in a network with a firewall at the edge.

[20]https://www.paloaltonetworks.com/cyberpedia/what-is-microsegmentation

4.3.4 DevOps Protection

DevOps is the combination of development and IT-operations. These teams are part of an agile development. As microservices support the agile development the infrastructure and development cycle should also be secured. Each developer submits code to a Version Control System (VCS) where different versions of various developers get merged into a release. Modern VCS support the use of a CI and a CD which allow automated tests, image and package building and deployment to production environments. Both, inside the code and inside the CI/CD configuration may contain credentials placed by careless developers. Once inserted, it is very difficult[21] and sometimes not possible to delete the credentials inside a VCS. Most public VCS systems allow a direct connection to a cloud provider for direct deployment which makes them very convenient. According to the principle of security layers of section 4.1, neither the infrastructure of the cloud nor the code itself in the public VCS providers is under control of the organisation. Furthermore, when management decides to go open-source all versioned credentials go public as well, which leads to serious security issues[22]. Open-Source is increasing in popularity [13].

Continuous Integration is intended to automate compilation, building and testing of software [85]. CI also automates vulnerability checks and therefore increases software quality and security. Tampering with the CI compromises all code handled with this module and therefore a security concept for the CI is necessary. A big risk lies in the building and testing itself. As the CI needs to execute every command of the developer of all checked-in versions it is exposed to highly unstable, erroneous, and vulnerable code which appears during the development process [86]. This erroneous code needs to be executed in a sandboxed environment to limit escapes and resource consumption. Most CI provide access to the projects via a WebGUI. As this applies for any website, the WebGUI needs to withstand any website attack[23], which, when successful, provides attackers access to any developed code. Furthermore, attacks can address any step in the build process and always affects the CI as well as the resulting code, which is intended to become productive code [86]:

- *VCS checkout*: During this phase, the CI downloads the latest version of the code including external dependencies to build it. Malicious external content, as well as obfuscated referrals (e.g.: links in the code to internal resources) may compromise the process and resulting code.
- *Build preparations*: When preparing the building environment, the CI needs to configure it differently for every code project. This includes a varying set of commands which cannot be limited to a preset

[21] `https://git-scm.com/docs/git-filter-branch`
[22] `https://www.theregister.com/2020/12/16/solarwinds_github_password/`
[23] `https://owasp.org/www-project-top-ten/`

a closed set of commands. Executing the preparation with a least privilege principle, as well as automated analysis of the build steps are a good countermeasure against this threat.

- *Builder runs*: While building, the CI compiles and links libraries to the code. Also, all test artifacts are built. Attackers could insert malicious source code which will be executed during tests. These Remote Code Execution (RCE) include any kind of attack. A sophisticated attack is Thompson's trusting trust attack [87]: A compromised build server inserts malicious code into the source code and builds it. Trusted source code is inserted in the compromised build server, therefore compromised executables will be the output. These attacks are difficult to detect. Again, the usage of a least privilege principle is recommended to countermeasure attacks. Anti-virus systems can help, but are useless when zero-day attacks are executed.
- *Notifications*: Notification includes all build servers sending output generated by the build server to the master server. If an attacker compromised a build server, it can modify these notifications, sending corrupt logging messages to the master (similar to modifying any logging messages of compromised server).

As seen, isolating building processes (e.g.: sandbox or VM), analysing source code and configuration data, as well as assigning least needed privileges to building and testing processes are an effective countermeasure against attacks of the CI. More concepts should be developed individually in each environment.

Similar to CI, also the Continuous Deployment is an essential part of DevOps. CD allows to automate continuous and reliable deployment of already build software [43]. CI is only a part of the CD, as it includes the developer sending code to a code repository, the CI retrieving code from the repository and building and testing, and automated deployment to a productive server. This pipeline from the developer to the productive server needs to be secured, authenticated and authorised. Poorly secured access to various parts in the pipeline leads to code injection and compromised productive server. The usage of provided authentication schemes of used tools is highly recommended, as well as the usage of common security practices: public key authentication, encryption of communication channels, multiple perimeter concept (see subsection 4.3.5).

4.3.5 Security Perimeter Shift

Traditional security perimeters are placed along the system boundaries. All services inside the system are trusted. An attacker had to pass a strong security perimeter once, but once inside, an attacker could hop from service to service. Nowadays, this is considered insecure and insufficient [2]. Microservices need to communicate a lot with other, also previously unknown units. As these unknown, freshly spawned units can be malicious (see: subsection 4.1.6), "trust no one" is the name of the game. Therefore the concept of zerotrust[24] [88] should be implemented. Figure 4.3 illustrates such a defense in depth behaviour. Traditional security approach (1.) follow the principle “System boundary = Security perimeter”. The modern approach (2.) shifts the security perimeter directly to the services. All services protect themselves and do not trust any other service until authenticated. This is called zerotrust. The proposed approach (3.) follow a multiple perimeter concept. This includes zerotrust and at least one traditional perimeter at the system boundary. When implementing zerotrust, end-to-end encryption should be used when communicating. Although this is a modern approach to security, it cannot be fully trusted, as manufacturers tend to program software to bypass local firewalls[25] which can also be used by malware. Nevertheless, zerotrust is not only firewalling but authenticating and authorizing which can also be programmed native. The proposed approach of Yarygina *et al.*, zerotrust, is therefore extended by a mixture of both approaches, traditional and zerotrust as part of a second or multiple parameter concept. This includes at least one traditional security perimeter at system boundary combined with zerotrust. The various perimeters do not correspond to each other and act completely independent. If one perimeter gets corrupted, the other one is not. Additionally, there can be multiple perimeters around subsystems. The more farther away the system is from the perimeter, it is more likely to block unfocused attacks and noise. The nearer a perimeter control is to the service, the more likely it is to detect focused attacks, as well as rogue services. This enhances visibility of security assets.

4.3.6 API Protection

The shift of security perimeters does not mean equalising security within services. As also been applied in monolithic services, APIs exposed to the internet should be reduced to their absolute minimum and reduce the theoretically available control flow for attackers [14]. An API is an interface providing control of a service to another service. The more services are placed near the system boundary, the more strict hardening these services need. Also, an API exposed to the public needs an additional layer of security, as compared to APIs only exposed to the internal network. A system boundary perimeter could help reduce or remove noise on external interfaces. All misbehaving services inside the system boundary could be seen as compromised. This can either be done in a granular way or via access profiles, as seen in Figure 4.4. Services offering valuable and sensitive information and access should be placed more inside a system, guarded and only able to be accessed by more hardened services. Also more resources should be spent securing sensitive services, than on other services.

[24]`https://msandbu.org/demystify-zero-trust-design-never-trust-always-verify/`

[25]`https://arstechnica.com/gadgets/2020/11/apple-lets-some-big-sur-network-traffic-bypass-firewalls/`

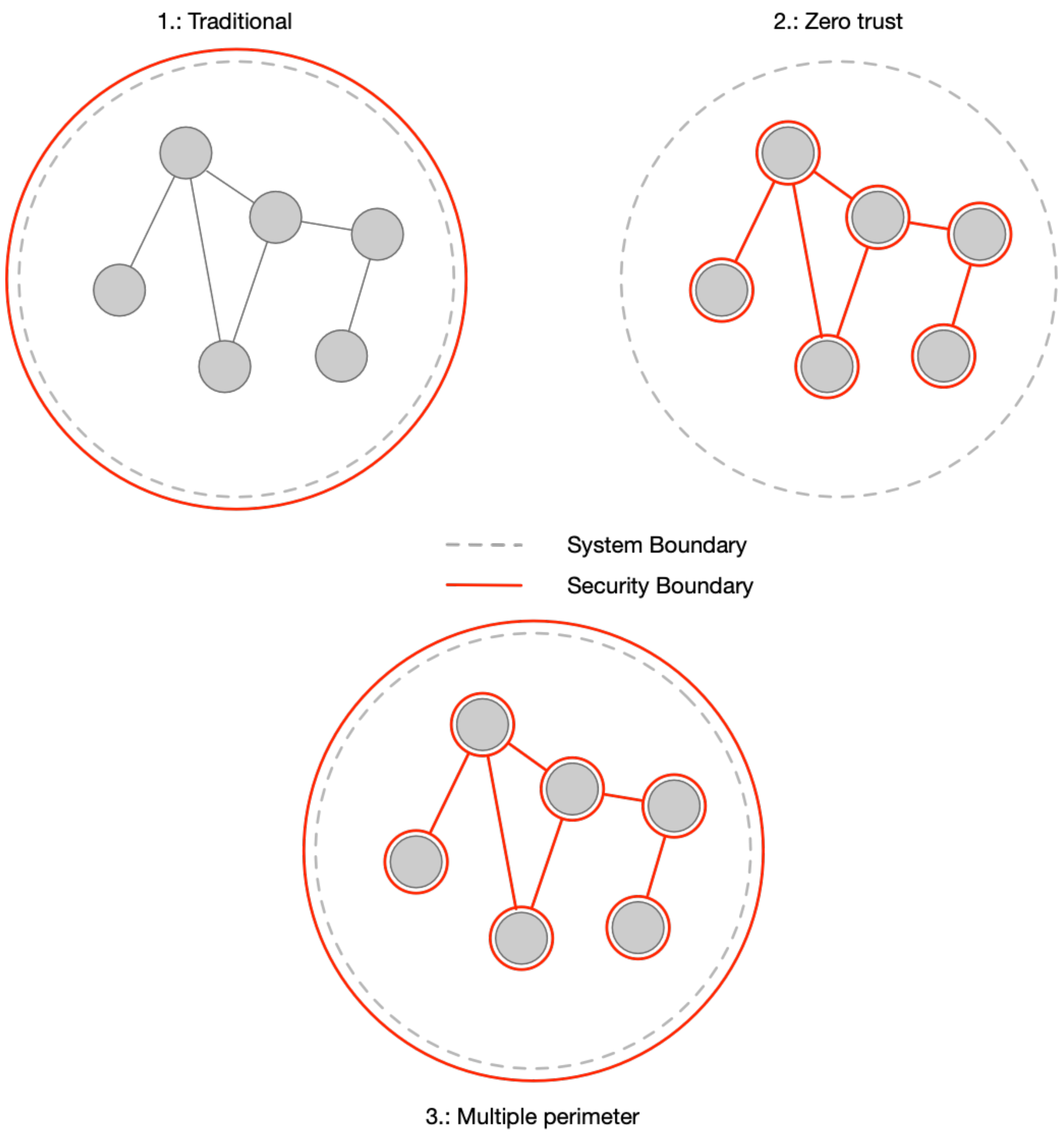

Figure 4.3: Differences between the traditional security perimeter, zero trust and multiple perimeter

REST use a set of architectural principles that help standardise APIs [13], but not secure them [89]. A common misunderstanding is, that REST secures APIs, which it does not. In fact, it is impossible to operate secure RESTful APIs [89], although, Fielding states that "REST emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems" [90]. The claim *enforce*

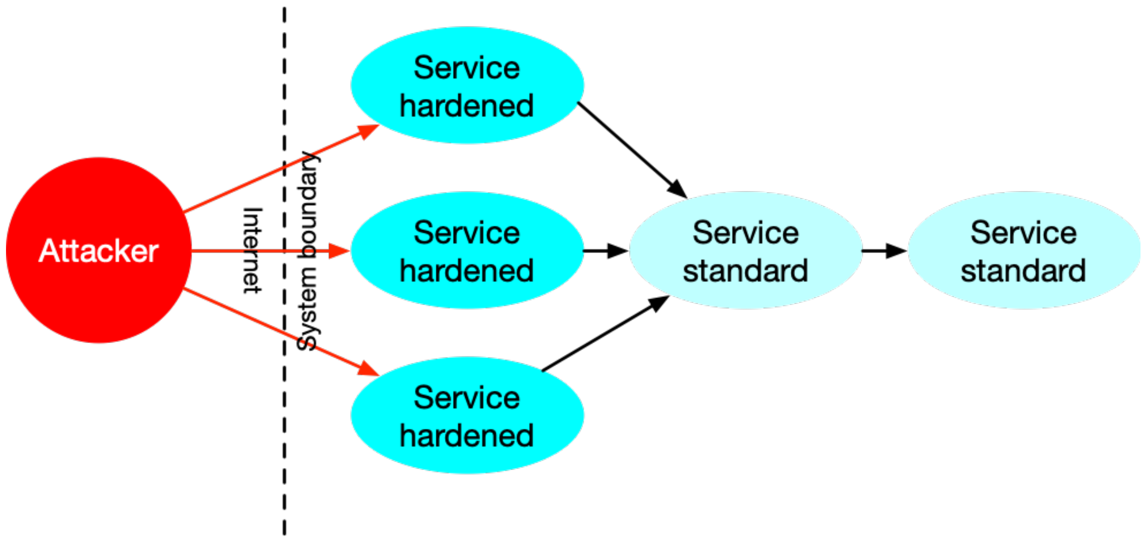

Figure 4.4: Asymmetric unit strength - Services near to the system boundary, offering APIs to the public, need a better hardening profile, than services only communicating with other services inside the system boundary.

security is not specified, neither in the dissertation, nor in other literature [89]. Yarygina observed three constraints regarding security in the REST architecture: stateless resource, caching and code-on-demand [89].

Stateless resource constraint: "The more security critical a system is, the more resource states it is likely to have" [89]. Token based authentication is the method that fits best for stateless authentication but is still not stateless. It is mentioned that the definition "each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server" [90] does not leave room for any information being stored on a server and therefore no room for security [89]. Security relies on information known to participating parties which is not possible in a strict stateless design. No replay attacks can be mitigated, as well as no nonces, counters or timestamps can be used to establish cryptography. No server keys could be used to sign or encrypt requests. APIs should therefore not be build RESTful, but only REST-like.

Caching constraint: The more dynamic and secure private content is transferred the less important caching gets. Dynamic content is not useful to be cached, as it is dynamic. Secure and private traffic is not wanted to be cached as it contains information that needs to be forgotten and not stored by further parties. This makes the caching and its constraint non-relevant.

Code-on-demand constraint: Client script languages like JavaScript are sent to a client and execute code at the client. This could be malicious code. The biggest concerns are, that clients cannot authenticate received code and cannot limit the scope of the code. Some security mechanisms like Data Execution Prevention

(DEP), Address Space Layout Randomisation (ASLR), and sandboxing are helping to mitigate damage from malicious code, but do not solve the problem entirely.
Another approach is the Enterprise Service Bus (ESB) or Message Oriented Middleware (MOM), which already introduce some security [26].

4.3.7 Input Validation

Input validation is the most crucial thing when communicating with untrusted entities. Unsuccessful input validation leads to injection of malicious code or data into your system[26]. Input validation is crucial for all input based applications, monolithic services and microservices. Microservices communicate a lot over the network, which is not trusted, to other entities, which are not trusted (when applying the zerotrust principle; see subsection 4.3.5), therefore a lot more effort in input validation must be made. APIs mostly use standardised protocols (e.g.: Hypertext Transport Protocol (HTTP), JavaScript Object Notation (JSON), Extensible Markup Language (XML), SOAP) for communication and input [91] [92] [93]. Implementing a REST API would help mitigate a lot of architectural failures [13].

4.3.8 Stay Simple

As explained in subsection 2.1.3 microservices should follow the guideline “Do one thing and do it well”. When using a DDD (not to be mixed up with Deadline Driven Development) the code base stays relatively small, which results in little lines of code (LOC) [2]. LOC are statistically correlated to bugs [94]. More LOC result in more bugs. More bugs result in more exploitable bugs. Therefore a smaller code base results in a smaller attack surface. Generally, also less code is easier to maintain and leads to a better understanding of the code to developers. Nevertheless, system architects still need to keep an understanding of the global system.

4.3.9 Package Dependency

Most software is greatly dependent on poorly maintained software, which was never questioned [95]. Dependencies are used, because software projects are getting more complex and developers do not have the time and resources to write every piece of code themselves, especially when this software already exists.
As the software may not be maintained anymore, or contain unpatched security flaws from the beginning, using it creates a security flaw in itself[27]. Container [96] and community library dependent programming

[26]https://owasp.org/www-project-top-ten/2017/A1_2017-Injection
[27]https://dlorenc.medium.com/whos-at-the-helm-1101c37bf0f1

languages like python [97] are especially vulnerable to dependency flaws. To address the patch problem, a monitoring of used dependencies is useful. Dependabot is a tool, that notifies when new versions of dependencies are available, and can automatically create new pull or merge requests to a VCS repository. When properly set up, a pipeline can automatically create, test and deploy the newly created software with the updated dependency to production. Also, monitoring is needed to detect code which is not receiving any update anymore. This dependency should be replaced with an active developer community. The monitoring also needs to catch security advisories of software. When a security relevant bug has been found such an advisory is sent out, telling users to either update their software version, or it contains a work-around if no update still exists.

4.3.10 Loose Coupling

Microservices should apply the approach of loose coupling. The services should be as much independent from other services as possible. The coupling can be measured through various metrics [26]:

- Space decoupling: Interacting parties are not directly connected to each other. They use an intermediary to refer to each other.
- Time decoupling: The individual units do not need to be available at the same time.
- Synchronisation decoupling: The interacting parties are still available while interacting.

When decoupling services of space, time, and synchronisation interface coupling gets stronger. How strongly a service is coupled to other services is measured via three metrics [26]:

- Number of interfaces: The more dependencies a service has, the more interfaces it has.
- Frequency of interface use: The more frequent an interface is called, the more dependent a service is from another one. Merging these services could be a solution to resolve this dependency.
- Interface evolution: Fast changing services result also in fast changing interfaces. When upgrading an interface to a newer version, it may break backwards compatibility.

The various interface methods address various coupling metrics. Table 4.3 displays the coherences. REST does not enable decoupling, only partially for the synchronisation decoupling, but only leads to low complexity. Messaging enables to decouple services, but require a complex infrastructure and a complex service.

4.3.11 Automation

Microservices should, as any other service, be built in a secure and stable way. As part of this, the system itself also needs to be stable and secure. When a node stops working or gets corrupted, the system

Interaction Paradigm	Decoupling			Complexity	
	Space	Time	ASync	Infrastructure	Service
REST	No	No	Producer	Low	Low
Messaging	Yes	Yes	Yes	High	High

Table 4.3: REST and Messaging compared about decoupling and complexity of implementation [26].

needs to react to this condition and heal itself. A new node should be built to replace the old one. To motivate developers to regard this situation Chaos engineering[28] like Netflix' ChaosMonkey are set in place. ChaosMonkey kills units randomly to encourage developers as well as system engineers and administrators to automate node recovery. This automated establishment of new units has, beside system stability, two positive side effects: introduction to permanent code changes without downtime and increased security by mitigating persistence [2]. When regularly deploying units and replacing old ones, employees are used to regular changes and introduced automation to this process which enables the introduction of new code easy to existing infrastructure. Also through the permanent rebuilding, attackers which inserted themselves in a node or introduced a malicious node will get kicked out of the system as soon as the malicious node will get replaced with a new one. Note that attackers that already hacked a system can do this again if nothing gets changed. So a node rebuilding only removes attackers temporarily. Also, if multiple units are affected, the attacker can spread from old units to new ones like a virus, because only a few units are rebuilt at a time, which gives time to the attacker to recover.

4.3.12 Isolating Units

To limit damage, when units get compromised, microservice units should get isolated as much as possible from other services. This means only letting units communicate with services they are intended to communicate with, and to only let them share data which is necessary: "share nothing principle and strict data owning" [2]. This principle correlates with the zerotrust approach (see subsection 4.3.5) and needs increased attention when when implementing automated deployment regularly (see subsection 4.3.11). When isolating units as described, with units getting killed and rebuilt, the risk of data loss is ubiquitous. Mirroring units could mitigate this risk. Also, microservice systems need to tolerate partial failures. The systems should detect outages and failure automatically and limit the propagation to prevent cascading failures [24] [27].

[28] `https://en.wikipedia.org/wiki/Chaos_engineering`

4.3.13 System Heterogeneity

Heterogeneous systems, meaning the usage of various OS, software stacks and libraries, mitigate low-level exploits [14] and require attackers to have a fundamental and embracive understanding of used systems to exploit all required owns [98]. This often narrows attackers down to big intelligence services due to lack of time, personnel and money resources of other attackers. Microservices support heterogeneous systems by design (see subsection 2.2.1). Otterstad *et al.* suggest to combine isolated microservices and diverse software to increase security [14]. As visualised in Figure 4.5, they identify three different fundamental attack phases compromising microservice systems: The initial attack, the sandbox escape and the lateral exploit trying to compromise the remaining existing system. The initial attack compromises a service. This service is used as a host to escape the sandbox, e.g. a hypervisor, and compromise the upper layer via the second attack. Once the hypervisor is controlled, all layers below are also viewed as controlled (see section 4.1). These means once a sandbox escape succeeds, the complete machine with all virtual machines, containers, etc. on this machine is compromised. The third attack is the lateral attack. This attack is executed from a controlled host and tries to compromise a host on the same layer. It is desired to compromise services on other machines to further distribute through the network.

System heterogeneity has the goal to snooker an attacker and provide a defender with more time [14]. System heterogeneity is used to make attacks statistically less efficient and therefore requires an attacker of the knowledge of multiple systems and exploits. Using several OS, various programming languages, multiple cloud providers, hypervisors, compilers, ASLR versions, and different backend services helps mitigating these attacks and are most likely a big impact when limiting successful attacks to a limited set of services.

4.3.14 Reduce Inter-Service Communication

Microservices should reduce their communication interfaces to a minimum. Only relevant information should be exchanged. Also the relationship between services needs to be a minimum [14]. As seen in Figure 4.6 service A can communicate to service B and service B can communicate to service C, service A does not need to communicate to service C directly because it could communicate to it via service B. This results in a lower performance and more complexity but also in a lower attack surface. This statement is in fact disputed. More complexity leads to more LOC and therefore to more bugs (see subsection 4.3.8) and it reduces the code comprehension (see section 2.2.1).

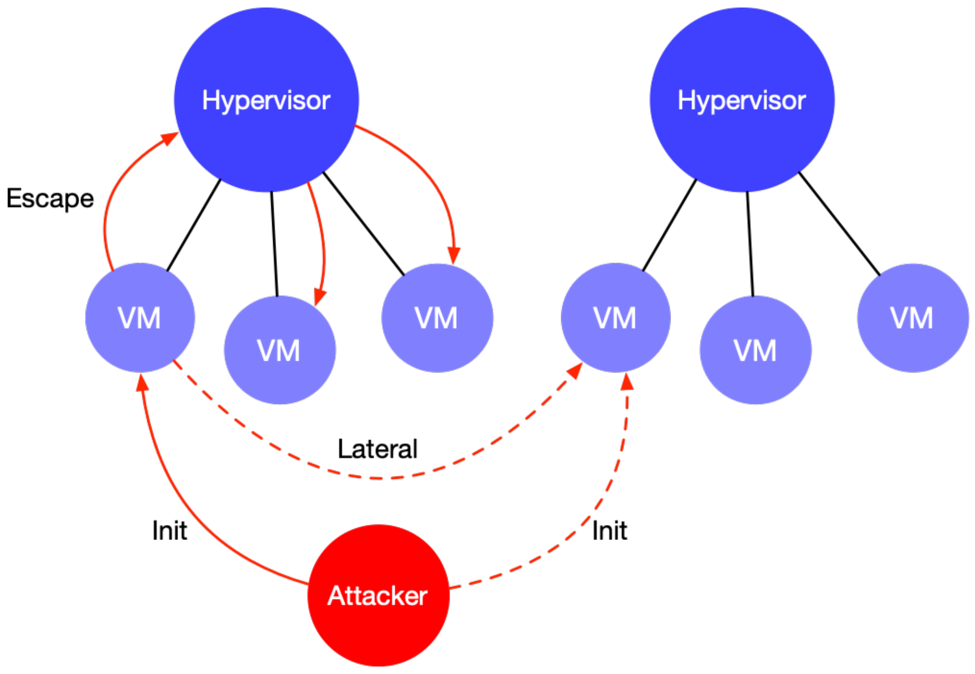

Figure 4.5: The three different attack phases when compromising a modern system.

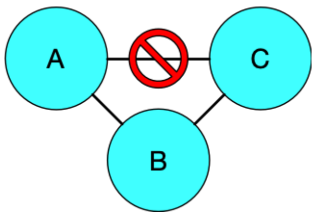

Figure 4.6: As service A can communicate to service C over service B, it does not need a direct connection to service C.

4.3.15 Securing the Service Mesh

“Service meshes have emerged as an attractive DevOps solution for collecting, managing, and coordinating microservice deployments” [46]. Service meshes allow the separation of microservice relationships from the service code into configuration files. Service mesh tools allow to deploy microservices automatically once the required dependencies are available. Service meshes automate service discoveries and traffic flow management[29]. Service Meshes can be seen as an extended container orchestration tool. The service mesh is therefore a critical infrastructure component which represents a lucrative attack goal.

As the service mesh is an application in a microservice environment, most standard application and microservice security measures need to be applied. This includes securing the application code and binary

[29]https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh

stack, as well as securing the network stack. Hahn *et al.* explicitly name message encryption, network ACL [46]. Furthermore they recommend to treat configuration sets as secrets, so only authorised personnel and services is able to view and edit things.

4.4 Applying Security to Microservices

The last section described some fundamental thoughts on microservice security, which differ from monolithic services, although microservice security thoughts are already influencing overall security thoughts, including monolithic services. The following section defines some best practices which should be implemented when working with microservices. Most of the following practices derive from the zerotrust principle (see subsection 4.3.5).

4.4.1 Service Authentication via MTLS

The usage of Mutual Transport Layer Security (MTLS) with an own PKI is highly recommended. A TLS connection means authentication of one party, while MTLS means authentication of all participating parties. A PKI is build and automatically assigning certificates to new units to identify them. The given certificates can be used to differentiate different kinds of units (functional, management, testing, etc.)[30]. Also these certificates are used to authenticate units to each other and encrypt traffic between them. The issuing process works as followed [2]:

1. A new Certificate Authority (CA) with a key pair is created.
2. A hash of the root CA and a random secret is packed into a token which is provided to all units when deployed.
3. The node generates a Certificate Signing Request (CSR) using the token and submits it to the issuing CA.
4. The CA verifies the identity of the node and issues a certificate to it.

It is highly recommended to automate this process, as it enables to frequent exchange of certificates (mostly between two to three months). Also, not all units should change their certificate simultaneously. Docker Swarm also enables to rotate the issuing CA certificate. To connect to the CA the units establish a TLS connection (the CA already has a certificate to identify itself), when connection to other units, the units now can authenticate themselves and establish a MTLS connection (see Figure 4.7). A second way of providing certificates to microservice units is the usage of long and short living certificates [99]. The long living

[30] `https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki/`

certificates are stored into a security module (e.g.: TPM) and used to request short living certificates. These short living certificates are used for all the communication except for the certificate issuing. Issuing short living certificates mitigates the problem of certification revocation [100].

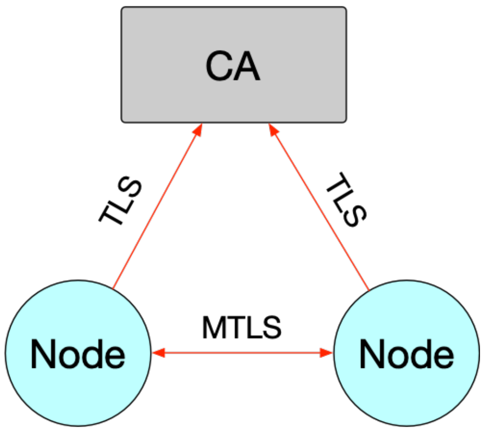

Figure 4.7: A trusted CAs issue certificates to containers to let them communicate over MTLS.

4.4.2 User Authentication via Token

Microservices will most certainly interact with user queries. This requires them to know the user and, more important, the state and authorization of the user query. Microservices should know if a user query was authenticated and the users role in terms of authorization [2]. Also every service should do this on their own, as microservices tend to hand over requests to other microservices. An often used example of this method is the propagation of requests with authentication and authorization via tokens in HTTP cookies. As cookies are client saved and handled entities, they cannot be trusted and must be handled with care [101]. Other standards have evolved to be around the JSON standards; JSON Web Signatur (JWS), JSON Web Encryption (JWE) and JSON Web Token (JWT).
Yarygina *et al.* present an own token architecture which will be presented in the following [2]. In general token-based architecture allows the propagation of a request in a secure and decentralized manner. They cause an extra service, a reverse Security Token Service (STS), generating security tokens for internal usage. They introduce the service to limit the expiration time of the token in its body. This behaviour introduces new vulnerabilities as malicious units can introduce false tokens with a higher expiration time than expected. The desired behaviour should be that all units know the maximum expiration time. As seen in Figure 4.8 token validation is a mandatory first step. It visualises a generic token based authentication. (1) A user request enters the API Gateway and gets redirected (2) to the user authentication service. The service authenticates the request and (3) requests a security token for this request (4) which gets returned by the reverse STS. The

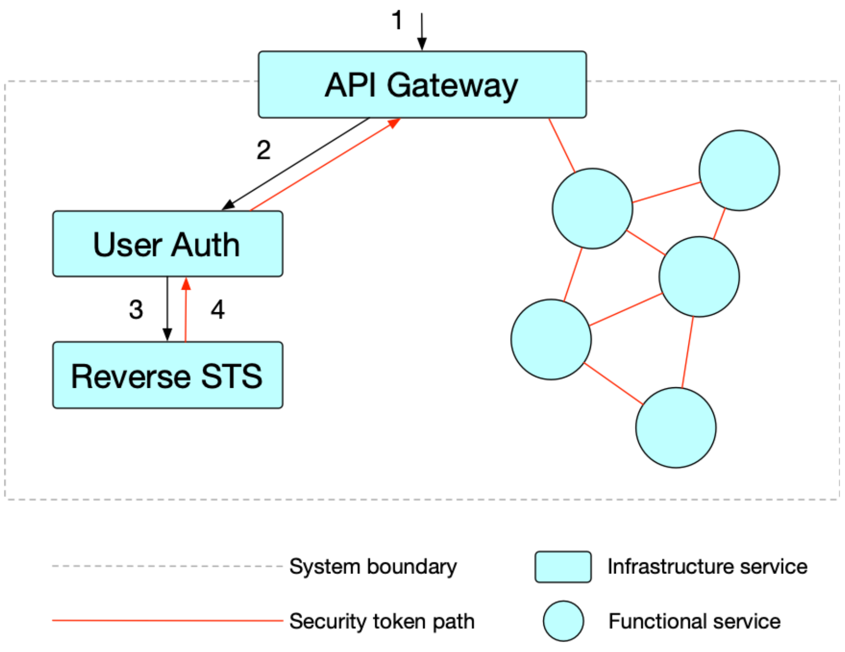

Figure 4.8: A generic token based authentication.

request is passed back to the API gateway along with the token, which passes the request and token through the functional services. This token is not given to the user. The security token in this example is not given to the user, but stays in the system. This increases security as well as system load. If a generic security token were generated and given to the user, this token could be used to authenticate the user in a given time frame (depending on the expiration time of the security token) without the need of contacting the reverse STS again. Also a security token given to the user should only include a timestamp which will be used to calculate the expiration by an owned service. This fits the second principle of designing distributed systems: A service will make decisions based on local information [24]. Already existing protocol standards as OAuth2.0 and OIDC can be used to delivery inter service security through tokens. Token based authentication methods need $O(1)$ server side to process N users which provides a high scalability [89].

To reduce traffic to authentication and authorization services, the security token could be given to a user for further usage. This token must include a valid time frame and an authorized context. After initially getting a token, the user directly requests a functional service via an API without the need to permanently request the authentication and authorisation services. This could be accomplished via JWS. A signed token containing already encountered data needs to be validated from the resource service. It will check the signature and grant the user resources based on the described rights. This approach requires the functional services to know the issuing certificate. Automatic enrollment is recommended. This token must have a short validity, around ten minutes, to limit damage, if the token gets misused in any way (stealing, hijacking, MITM, etc.). CRL will not scale to such an amount of issued certificates [100]. Figure 4.9 represents such an architecture. It shows a signed-token authentication and authorisation. (1) The user sends a login request to the gateway. (2) The user auth server authenticates and authorises the request, creates a token and (3) tells the CA to sign it. (4) The CA returns the signed token to the user auth server which returns it (5) via the API gateway (6) to the user. Now the user can request resources directly (7) via the gateway (8-18) from functional services as long as the signed token is valid, without the need of requesting the user auth again. Each service validates the token itself via the signature of the token. Requests 9 to 16 represent requests between functional services and are not printed for visibility reasons.

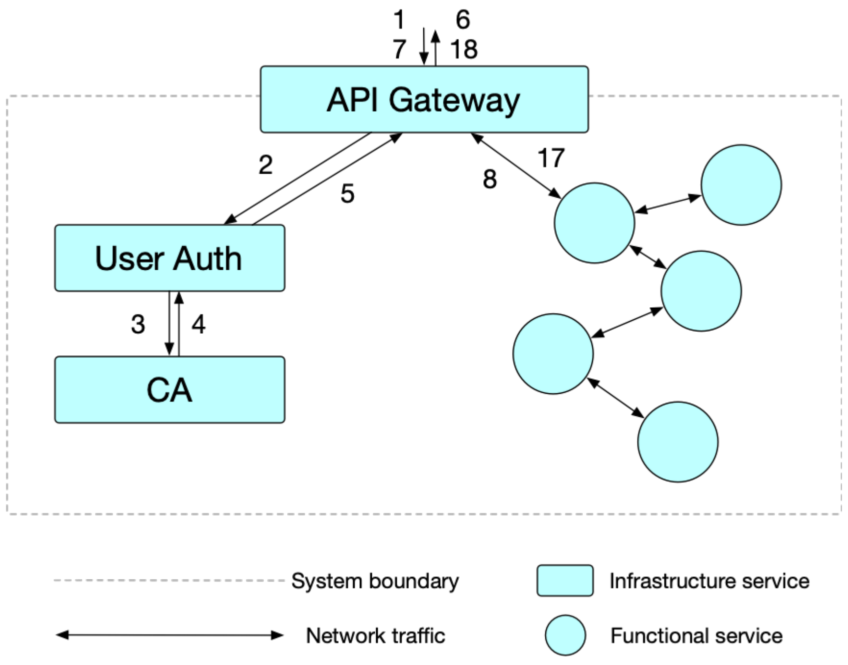

Figure 4.9: An Authentication and Authorisation service using signed token.

A big problem in these approaches is the clock synchronisation problem in distributed systems [102]. Time in distributed systems must be synchronised correctly, with the need of low resources, and should stay within a tolerance. Another problem is that these tokens are reusable. To protect them from replay attacks they should get transmitted through secured connections only to trusted peers, e.g.: a MTLS connection. This is highly risky why the token approach should be changed to a challenge-response approach. A third problem, as in any asynchronous cryptography, is the secrecy of the private key. As soon as the private key is known to an attacker, the system is to be seen as compromised.

4.4.3 Ticket based authentication

A ticket based authentication like Kerberos[31] is not recommended in a microservice context. A Kerberos needs a Ticket Granting Server (TGS), an Authentication Server (AS) and a Service Server (SS) to work. The TGS and the AS need a shared storage to access a key database. This database contains all client and server keys which were used to encrypt granted tickets. Also Kerberos relies a lot on a very good clock synchronisation of all participants. To synchronise time in a network, Network Time Protocol (NTP) is used. It tries to eliminate time differences caused by participants, physical distances between participants and used physical network carrier. Synchronising time in a distributed system brings a lot of problems and does not scale well [102]. The described problems are simply limited to scalability, not security.

4.4.4 Service Authorization

Authorization of services is crucial to limit access of the services to their intended behaviour. Digital certificates could be used to authorize services [103] [104]. This is achieved by creating a multi-hierarchy PKI [33]. One root CA is accountable for all issued certificates in this system, granting authentication. A level below, every microservice type has its own issuing CA providing authorization, which can be named as realm. Microservices define which realm is allowed to communicate with by adding issuing CAs to their trust list. To allow access from all services, simply add the Root CA to the trust list. This approach is visualized in Figure 4.10. The Root CA issues certificates to its issuing CAs but never to other services (each CA can be built as a microservice itself). The issuing CAs issue certificates to their assigned services in their realms. The services define which service realm is authorized to communicate with it. Additional trust levels are achieved by introducing further CAs (Functional A CA issues a signing certificate to a functional A.1 service and functional A.2 service CA). As an example, the management services accept connections from other management services, but not from functional services. Functional A services accept connections from functional B services, but not vice-versa. As all certificates have a limited expiration time and as well certificates, as well as CAs need to be renewed on a regular basis, it is highly recommended to automate such a configuration.

4.4.5 User Authorization

To authorize users, multiple methods have been developed. Most common, modern approaches are Role Based Access Control (RBAC) and Attribute Based Access Control (ABAC). Yarygina *et al.* propose to use

[31]`https://web.mit.edu/kerberos/`

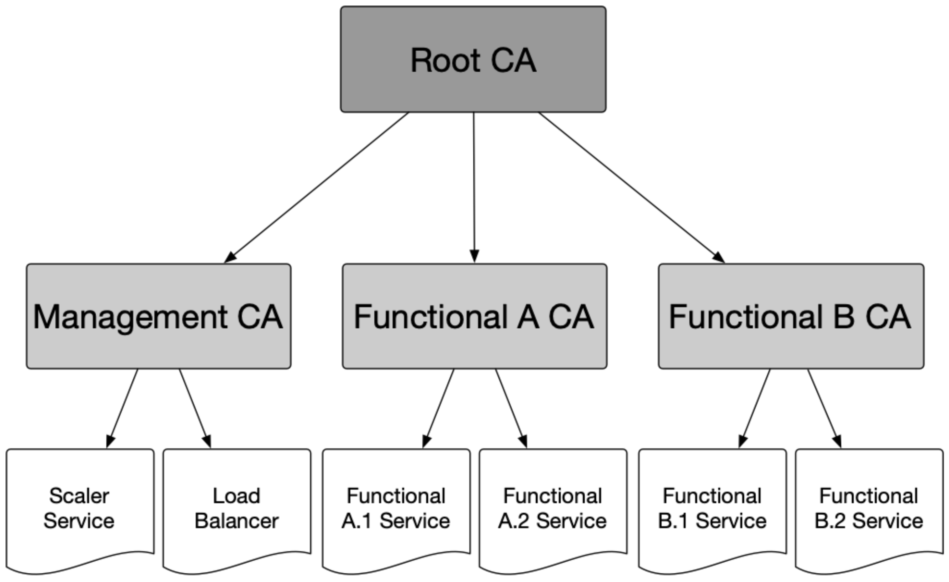

Figure 4.10: Service authorisation using Certificate Authority.

ABAC in microservice context to grant fine grained access to APIs [2]. Roles for RBAC could be transported via the security tokens of the authentication phase.

4.4.6 User authentication and authorization with existing frameworks

Already existing frameworks as OIDC[32] or OAuth[33] can be implemented in a service conveniently without the need to invent something new. These frameworks are also already widely adopted and tested for security and performance.

OAuth has two versions, OAuth 1.0 and OAuth 2.0. OAuth is used to authorize third party applications to an existing account. OAuth provides access to data of a user with the users permission to an application without the need of providing a password. Version 1 differs from version 2 as it still supports client request signing. As OAuth 2.0 is the successor of OAuth 1.0, the usage of the second version is strongly encouraged.

OIDC is the successor of OpenID and combines it with OAuth. OpenID was a provider of identities. A user could create an account at an OpenID provider and use this account to login everywhere else. With the combination of OAuth, also the authorization part is covered in OIDC. OIDC enables users to control every application compatible with a single account at an OIDC provider. These providers are referred to as Identity Provider (IdP) whereas the applications and webservices relying on the provider are referred to as Service Provider (SP).

[32]`https://openid.net/connect/`

[33]`https://www.oauth.com/`

Both, OAuth and OIDC rely on TLS for their confidentiality and integrity [89]. They provide scalability because they use server signed tokens to authenticate and authorize clients. Big companies already act as IdP, including Facebook (Login with Facebook), Twitter, Microsoft, Gitlab, Orcid, etc. When developing new services, the decision to use already existing accounts of one of these services can be made. Figure 4.11 may help decide the right authentication and authorization method. This decision tree helps to decide which authentication and authorization method is the right one for a service. It ends with either implementing OIDC as a consumer or a provider or developing an own method based on principles described in this chapter.

Security Assertion Markup Language (SAML) [105] is an older alternative from 2005 to OIDC based on SOAP and XML.

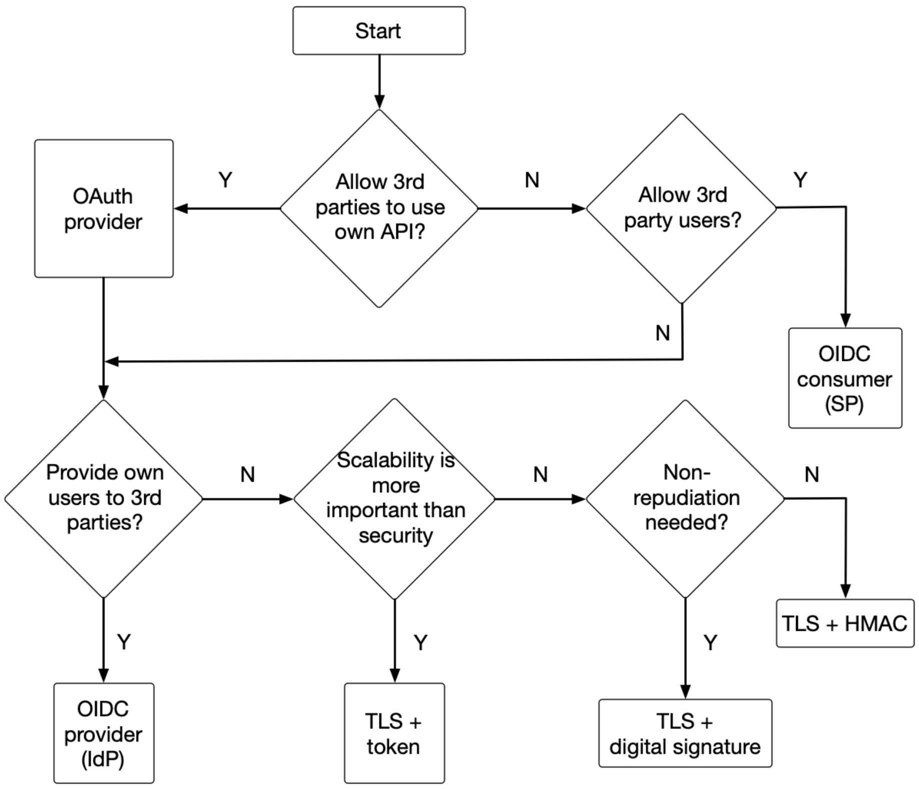

Figure 4.11: Decision tree to decide which implementation of user authentication and authorization should be used.

4.4.7 Security Monitoring

A security monitoring system watches the output of all units of the same service and looks out for anomalies [106]. Yarygina *et al.* propose a security monitor in form of a Machine-Learning (ML) game engine [107]. This engine works similar to an IDS [108]. Each microservice reports all inputs and outputs to the engine, also topology changes are reported [107]. The engine calculates a model of communication and adapts it with every topology change. Three different states can be assigned to each unit; *benign*, *under attack*, and *compromised*. Benign means everything is normal, under attack states that the unit is under attack at the moment, and compromised means a unit is suspected to be or is compromised by an attacker. The engine runs a minimax algorithm[34] from game theory to decide which actions to take. The following actions are allowed [107]:

- Restart and Rebuild: Restarting/rebuild the service with the same configuration will have the lowest impact on the system but also tends to fail to mitigate attacks. An attacker could be persistent on a service, or simply repeat the same attack again to gain control of a system. If the problem persists, a rollback to an older image of the unit may help.
- Recompilation and Rewriting: Mitigate attacks through heterogeneity and recompile or rewrite binaries with different arguments [14]. To rewrite a binary automated special compiler support is needed [109]. Recompiling effectively mitigates replay attacks.
- Diversification of cloud provider: Moving a service to another cloud provider mitigates low level attacks as this will imply usage of a diverse hardware and virtualisation software (see section 4.1). Also, it changes or removes the access of a cloud provider and enables another one to access the units.
- Scale up and down: Similar to *Recompilation and Rewriting*, several units of the same service but with different arguments are spawned. These services report their output to a master node which checks the output for anomalies. If a malicious unit is detected, it gets recompiled with new arguments. This is seen as self healing. Figure 4.12 shows such a behaviour. This action can also be implemented as a stand-alone solution without the game engine. The monitoring system watches the output of various services and notices anomalies in responses. The monitoring system is entitled to recompile the respective service with other compiler arguments and replace it with the new compiled image. The monitoring system can also be attached to the network like an IDS (e.g.: mirrored switch-port), therefore not be a single point of failure when it crashes.
- Split and merge services: A unit will be split at the function level. A binary attack would be mitigated, as previously existing paths were rebuild. This is a purely theoretical approach as the required tools for automated code modification do not exist yet.
- Isolation and Shutdown: Stops the service permanently. This approach is not applicable in high-availability environments. Noureddine *et al.* showed that permanently shutting down a service signif-

[34]`https://en.wikipedia.org/wiki/Minimax`

icantly delays an attacker [110].

Monolithic services allow to rollback, rollforward, isolate, reconfigure, and reinitialize the system [111], but, as seen, microservices allow more automated defense action. This game engine and the master node of *Scale up and down* need to become part of the infrastructure and orchestration.

Such a model has a much lower latency than humans when responding to alerts and a more objective and detailed view on a system [107]. As humans need time to resolve and address the issue and react, the game engine does it instantly. To address the resource usage of this complex algorithm, x will be the depth of the decision tree and y the count of legal moves in the minimax algorithm. Therefore time complexity results in $O(b^m)$ while space complexity results in $O(b * m)$. When sorting the results for the alpha-beta-pruning algorithm time complexity can be reduced to $O(b^{\frac{m}{2}})$. Also, a real world example will limit itself in granularity due to space and time complexity.

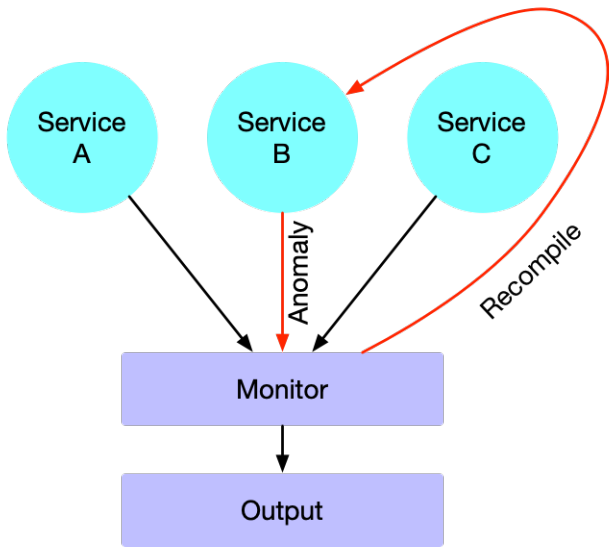

Figure 4.12: Automated low level security monitoring and response

4.4.8 Smart Endpoints and Dumb Pipes

As already described, the approach of loose coupling and decentralisation in microservices leads to a lot of network traffic. Therefore, there is a need for error detection and error correction of network traffic in microservices. Lewis *et al.* propose control all network flow inside the services [112], therefore act similarly to the network approach of the entire internet. This approach could be implemented via REST, Enterprise Service Bus (ESB) or Message Oriented Middleware (MOM). MOM protocols establish a pipe to transmit messages. Messages are stored into a pipe and are processed one after another using the First In - First

Out (FIFO) principle. Senders aren't awaiting complex return values but simple acknowledgments that a message has been received. The result will be returned with another pipe after completion. Common implementations are *Apache Kafka*[35] and *RabbitMQ*[36]. MOM and ESB does not work with HTTP messages, as in a REST architecture, therefore implementations of the chosen protocol should be available in every used OS and programming language. As already described in subsection 4.3.6 REST lacks security, therefore MOM is recommended. Common MOM implementations support MTLS for authentication and traffic encryption out of the box [26]. Nevertheless, when establishing system heterogeneity (see subsection 4.3.13) it comes with decreased performance.

4.5 Privacy Concerns of Microservices

Privacy is a topic not clearly described often. Perra describe privacy as the state or condition of hiding the presence or view [113]. As microservices are often deployed in a cloud environment, several privacy issues arise (see subsection 4.1.3). The following sections explain privacy problems of microservices in a cloud context and privacy problems of the microservice architecture itself

4.5.1 Cloud Privacy Concerns

Privacy concerns of the cloud are a barrier in adopting microservices [114]. As the advantages of microservices and other marketing measures increased the push of customers into the cloud, the privacy issues of the cloud are still unsolved. There is still a need for the storage of confidential things somewhere. "[...] privacy is needed to attain the data, user identity and controls" [13]. As microservice systems often include multiple IdP and SP where a lot of sensitive data is transferred, there is definitely a need for privacy [47]. Cloud privacy is a huge and complex topic, which will not be discussed in detail in this work (see subsection 4.1.3). The marketing teams of cloud providers help out, offering encryption in some formats, but this does not solve the problem of privacy, where cloud providers should not know that some data even exists. Also, a common misconfiguration of cloud encryption is to store the used key in the cloud storage [115]. This is often chosen because it is convenient, but it enables cloud providers to decrypt the data.

The author of this thesis proposes to split data and calculation to various cloud providers and only combine them at on premise hardware. Therefore cloud providers do not know the full data and calculation. The data and stored and the calculations executed in one cloud environment can be seen as garbage, as they

[35]`https://kafka.apache.org/`

[36]`https://www.rabbitmq.com/`

only add up when correlated and encrypted at the own data center with a key stored at the own data center. This enables to use the advantages of rapid scaling of cloud infrastructure and still preserves privacy and enhances security.

If this approach is applicable in the field and which pitfalls need to be regarded is unclear. This is why further research definitely needs to happen in this field, which can also result in the fundamental failure of privacy in a cloud environment.

4.5.2 Architectural Privacy Problems

Microservices themselves also have a privacy problem which has not been researched well. Vistbakka *et al.* tried to specify the problem via an example in the medical fields [29]. A patient gets diagnosed with a disease in hospital A. This hospital stores the patient's data and medical records in it's own datacenter. The hospital advices the patient to consult a different hospital B for a cure. To make this more clear, we assume that, Hospital A is in the home country, hospital B is abroad. To cure the disease, Hospital B needs the medical records of the patient and receives them from the datacenter of hospital A. Also, hospital A forwards the data to other data consumer, such as insurance agencies and pharmaceutical companies. This behaviour is visualised in Figure 4.13. It also shows non-authorized access to this data as an example. Analogously to the physical world, when the patient give data to hospital A and it redirects it to a data center, neither the patient nor hospital A own this data anymore and cannot control where it goes. Monolithic programs have less problems in handling data than microservices do. The data-intensive, distributed nature of microservices has an explicit problem with data privacy as a service does not control the data once it passes it to another one. As the data transfers in this example do not mean that these privacy violations are intentional, this microservice architecture cannot monitor the satisfaction of privacy constraints and cannot prevent a violation of them. Each patient, and furthermore each user and service, must be able to decide to whom and where to share data and microservices as the tool must ensure that these guidelines are executed.

To make a first step and address this problem, Vistbakka *et al.* defined a privacy preserving microservice system [40]. They defined microservices privacy-preserving if they only process data they are entitled to and it they only share data, which the recipient is entitled to process. (Service A only uses and requests data which it is allowed to use and only shares data with service B which service B is allowed to use. A complex formal description is found in the cited paper [40]. Overall, this definition could easily be adapted to match every service, business process and even human behaviour.

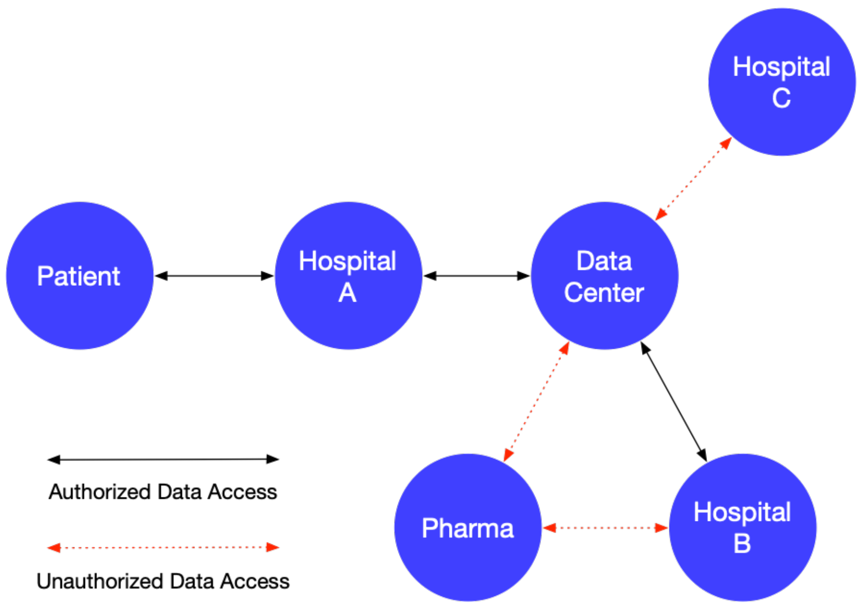

Figure 4.13: Data sharing in the medical fields as an example, visualising authorised and unauthorised data access.

4.6 Decision helping guide

Finally, we present a guideline, if one should use microservice or monolithic services, as well as a guideline on how to secure microservice, based on the challenges and solutions already described.

4.6.1 Monolithic service or microservice?

The decision to create a microservice should be deliberated carefully. Why is there a need to create a microservice instead of a monolithic service? There are two different types of created microservices; the ones that are originally created as microservice and the ones that are transformed from monolithic services. To decide if one should create a microservice from scratch, the following assets and properties should be answered:

- Scalability: Microservices are all about scalabilty. Is there a need to scale services up and down on a frequent, irregular basis. Is there no possibility to let a service idle due to limited resources or financial thoughts?
- Heterogeneity: Is there a need for several OS distributions or programming languages?
- No on premise: The offered service is a hosted service, not software to sell. Usually this means, the company sells PaaS or SaaS services. The customers will be forced into a cloud aspect and has no possibility to host this service themselves.

If these questions are positively answered, a need for microservices can be seen. Scalability is still the strongest argument to adopt a microservice approach.

Bear in mind, that microservices need a lot more infrastructure in comparison to monolithic services [25] and need a specialised DevOps team to maintain the infrastructure. Creating microservices, means permanently maintaining an offered service.

When deciding to transform a former monolithic service into a microservice, it rises further security aspects. Monolithic services may include code and routines that were never meant to be public. When transforming them, these sensitive data and routines must be located and taken into account!

4.6.2 Checklist of microservice security

The following checklist should be a guideline on what needs to be considered when creating microservices. It is based on the proposed layered principle (see section 4.1). The first three layers, hardware, virtualisation, and cloud play a big role in the overall security and definitely needs to be considered as a business concept, but cannot be adjusted when creating microservices.

The first layer that is directly influenced by microservices is the *communication*. Microservices communicate a lot and therefore an established trust and secure communication is the key. Consider the following:

- Network isolation and microsegmentation: Limiting the network partners of microservices and limiting access to them greatly reduces the attack surface of a microservice.
- Security perimeter shift - zero trust: Combining the traditional perimeter security approach with the uprising zero trust principle greatly increases security of each service. Consider also, that zero trust relies on a correct configuration of all participating services.
- Reduce inter-service communication: Microservices should only communicate about the absolute minimum and should only rely on an absolute minimum of microservice relationships. Beware that this increases service complexity and contradicts the microservice privacy aspect.
- Usage of TLS and MTLS: The usage of transport encryption via TLS enables confidentiality and integrity between services and can be used as an authentication and authorisation mechanism between services.
- Usage of authentication and authorisation: Services must be authenticated and authorised. This can be accomplished via the usage of TLS, existing frameworks like OAuth or OIDC, or security token. Similar to services, users must be authenticated and authorised: OAuth or OIDC, or security tokens. The recommended way is the usage of TLS for confidentiality, integrity, and service authentication and authorisation as well as the usage of OIDC for user authentication and authorisation.

The next layer is about the *service* itself:

- Input validation: Microservices mostly rely on the container architecture, which shares resources on a node with other container. Also microservices communicate a lot over a network. Therefore no external loaded information, either over network or file system is a trusted source of data. Consequent input validation is the key.
- API protection: Microservices expose a lot of functionality to other services or the public. These exposed interfaces must be protected.
- Package dependency: Most modern programming languages rely on the usage of community libraries. Containered microservices even increase the fact, that some container images rely on other images. Therefore a lot of community code could be present in the resulting software, which contains already known flaws. A dependency monitoring and update management needs to be implemented.
- Loose Coupling and isolating units: Microservices should work on their on and only share data when absolutely necessary. They must not rely on shared infrastructure. In further consequences, microservices should be isolated from each other to limit possible damage, when being attacked.
- Heterogeneity: Microservices should make use of their relative ease of system and code heterogeneity. This principle could also be automated. Heterogeneity reduces the impact of low-level attacks.

The last layer is about *orchestration*:

- DevOps protection: The working environment of the DevOps team requires a proper security evaluation. Never store credentials in plain text, nor save them to VCS, in fact establish systems, where there would be no need to do so. Isolate the development and build environments from production systems. As automation is the driving principle in the DevOps area, these automation processes need to be secured and monitored to prevent intrusions into the software development and prevent the delivery of malicious software.
- Security monitoring: Monitor in production services and locate anomalies in their network traffic. Anomalies could be a sign of a compromised service. Automated recompiling and redeploying of this service could mitigate and revert a successful attack.

Overall, the concepts of the CIA triangle, confidentiality, integrity, and availability should lead the security thoughts. Also, the microservice itself should be simple and easy maintainable.

5 Conclusion

The main challenge, when designing a secure system, is to initially establish trust between the disparate services. As shown, there is no standardized way to establish security in microservices. Existing solutions are either closed source, or proprietary and therefore not portable to other solutions. In general these concepts are poorly documented and poorly understood. Also an independent, widespread performance cost overview is missing. Yarygina *et al.* implemented an own system and observed a performance decrease of 7% when using security tokens and a decrease of 4% when using MTLS as opposed to not using any security at all [2]. When taking into account that microservices can perform up to 80% lower than monolithic services on the same hardware [25], this is a relatively high value, so they propose reusing existing connections wherever possible. Yarygina states that in some cases the operational overhead of microservices is not acceptable and unnecessarily expensive and needs to be evaluated if a program really needs to be created with a microservice structure [26]. Also, often monolithic services are transformed into microservices without any need. Monolithic services are still useful as they have a completely different architecture and therefore perform differently. Used CAs and the token services should run in a hardened environment. Many authors suggest a TPM or SGX environment, even though there is no guaranty, that these are in place in a cloud environment. As seen in section 4.1, when applying microservices at cloud provider, the applied hardware security only relies on trust into the cloud provider, as there is no proof they establish these features. In general, critical services and services offering APIs to the public (subsection 4.3.5) should be hardened with an extra focus: The nearer the service is to the system boundary, the more security it needs.

As microservices communicate a lot over the network, a focus on network security should made. This complies with the challenge of establishing trust between microservices. Network isolation and a flexible Software Defined Network (SDN), as well as microsegmentation play an essential role. Additional microservices should communicate over a trusted and encrypted channel, using TLS or MTLS in combination with a self-hosted CA. Precautions needs to be made, as hosting an own CA is not a trivial thing to do (Lifetime of certificates, Issuing process of certificates, CRL policies, Security of the CA itself, etc.).

Analog to the saying “never do your own crypto”, the author recommends the usage of already existing, established authentication and authorisation protocols and implementation rather than creating a new one (OIDC; see subsection 4.4.6). When creating a new architecture a signed token approach (see subsection 4.4.2) is the most scalable solution.

As seen in subsection 4.3.4, most of DevOps related attacks, can be mitigated using the least privilege method when assigning user rights, isolating build processes and ensure confidentiality and integrity using encrypted communication channels, so therefore apply general security best practices.

System heterogeneity is a property that is not difficult to achieve when using microservices and helps a lot to mitigate low-level attacks. System heterogeneity in terms of automated infrastructure can even be automated up to a certain extend (see subsection 4.3.13 and subsection 4.4.7). System heterogeneity is often only regarded as system distribution heterogeneity. As already explained, Container technology only enables the variation of distributions over a shared kernel. Kernel heterogeneity increases complexity, as one node can only host one kernel with multiple container sharing the same kernel and the orchestration solution must be able to correspond with multiple kernel solutions.

Sensitive data handled by cloud microservices should be reduced to an absolute minimum, as cloud providers cannot guarantee any privacy in their environment (see subsection 4.1.3 and section 4.5). Privacy is defined as someone not knowing certain data exists.

Microservices reduce complexity of the services themselves but require a complex infrastructure to have some security features enabled. This infrastructure could be more complex than the infrastructure needed for monolithic services. As security in monolithic services often comes while programming the services, microservice security comes mostly with extra work that must be done. This leads to fewer security features or none at all enabled by default, as it will cost time and human resources to create security measures, but will not make any money directly.

Also, a study showed that the agile approach could decrease the time to market of new code to 20 minutes [116]. This incredibly fast deployment and the increasing numbers of automated attacks create a need to secure the testing and deployment road to avoid delivering erroneous or compromised code and to automate security concepts and features, and also, there emerges a need of self protection like it is done via the methods described in this works.

Overall, microservice security is a field not well researched. As there is no standard or approved guideline in building microservice systems, everyone is basically just working on how they think it works best, which is most likely a bad idea. This leads to architecture, that may undermine the benefits of choosing a microservice approach in the first place and to security issues, as only a few people objectively check this architecture and these security measurements. The main security challenges, different from monolithic security challenges are adoption of automation as an architecture, securing the communication, cloud concerns, as well as the distribution of services. As the software gets automated, also the security design and approach needs to strongly depend on automation. Communication between different services is a key part of a microservice architecture which needs to be secured. The distribution of services creates a need of an amount of authenticating and authorising various software components vice-versa in the system which is new to software development.

Also privacy in a microservice context has not been researched well yet, although a formal definition already exists (see subsection 4.5.2).

5.1 Future Work

Further to the already discussed topics, more research about protection of microservice orchestration as well as the service mesh needs to be done. As orchestration resp. service mesh is the control of the system, when being compromised, it enables attackers to compromise the whole system (see subsection 4.1.6).

Another statement was that microservice relationships should be reduced to a minimum (see subsection 4.3.14). As this increases complexity as more services are simply used as proxies, it therefore also increases LOC and errors being made (see subsection 4.3.8). There is also a greater dependency on other services. These are two paradox statements which need further research, if they keep themselves in balance or if one statement is not true or negligible in specific situations. Furthermore, it is against the privacy definition of Vistbakka *et al.* (see subsection 4.5.2), as services get in contact with data, they don't need to.

Otterstad *et al.* argue that microservices need to be spawned automatically with different compiler arguments (see subsection 4.3.13). These services should be monitored for anomalies and rebuild with different arguments when necessary (see subsection 4.4.7). As these services are not tested and are only monitored to watch how they perform in production, further research needs to be done if this really works and what should be best practices to achieve a high detection rate and to see which arguments must be used in which order to successful mitigate attacks.

Also, the rise of microservices and automated service deployment lead to the development of Software Defined Network and microsegmentation. The field is not new and widely used, but only little research has been done on how SDN affects network security and microservice security in particular.

Further research definitely needs to take place about privacy in a cloud context. Sensitive data is transferred to cloud storage and services, often on situation wherin the cloud providers should not even know that this data exists, but privacy in a cloud environment is not possible yet.

A big topic is the bad performance of microservices, compared to monolithic services; especially when some security is applied. This operational overhead needs to be resolved to increase security usage and acceptance in microservices.

Apart from the privacy aspects of the cloud, the privacy approach in microservices have not been researched well yet. At the time of writing this work, the DBLP[1] only returned four results, all in medical fields, when searching for *microservice privacy*. There is definitely a huge need for further research in the basics of this field.

Overall, there are a lot of introduced approaches to work with microservices, but there are still no guidelines or best practices. Most of the proposed approaches have not been surveyed yet. After around 13 years of microservice development, a guideline, best practice, or even a standard specification would help a lot to build secure microservices.

[1] `https://dblp.uni-trier.de`

List of Figures

List of Tables

Glossary

ABAC	Attribute Based Access Control
ACID	Atomicity, Consistency, Isolation, Durability
ACL	Access Control List
API	Application Programming Interface
AS	Authentication Server
ASLR	Address Space Layout Randomisation
CA	Certificate Authority
CD	Continuous Deployment
CDC	Cyber Defence Center
CI	Continuous Integration
CPU	Central Processing Unit
CRL	Certificate Revocation List
CSR	Certificate Signing Request
DDD	Domain Driven Design
DDoS	Distributed Denial of Service
DEP	Data Execution Prevention
DMZ	Demilitarised Zone
DoS	Denial of Service
ESB	Enterprise Service Bus

FIFO	First In - First Out
HIP	Host Identification Protocol
HMAC	Hash-based Message Authentication Code
HSM	Hardware Security Module
HTTP	Hypertext Transport Protocol
IaaS	Infrastructure as a Service
IdP	Identity Provider
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
JSON	JavaScript Object Notation
JWE	JSON Web Encryption
JWS	JSON Web Signatur
JWT	JSON Web Token
LOC	lines of code
MITM	Man-in-the-middle
ML	Machine-Learning
MOM	Message Oriented Middleware
MTLS	Mutual Transport Layer Security
NIC	Network Interface Controller
NTP	Network Time Protocol
OIDC	OpenID Connect
OS	Operating System

OSI	Open Systems Interconnection
PaaS	Platform as a Service
PKI	Public Key Infrastructure
RAM	Random Access Memory
RBAC	Role Based Access Control
RCE	Remote Code Execution
REST	Representational State Transfer
RNG	Random Number Generator
RP	Relying Party
SaaS	Software as a Service
SAML	Security Assertion Markup Language
SDN	Software Defined Network
SGX	Software Guard Extensions
SIEM	Security Information and Event Management
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SP	Service Provider
SS	Service Server
SSL	Secure Socket Layer
SSO	Single Sign On
STS	Security Token Service
TGS	Ticket Granting Server
TLS	Transport Layer Security
TPM	Trusted Platform Module

UI	User Interface
VCS	Version Control System
VLAN	Virtual Local Area Network
VM	Virtual Machine
VPN	Virtual Private Network
XML	Extensible Markup Language
XSS	Cross-Site-Scripting

Bibliography

[1] Nicolai M Josuttis, *SOA in practice: the art of distributed system design.* " O'Reilly Media, Inc.", 2007.

[2] T. Yarygina and A. H. Bagge, "Overcoming security challenges in microservice architectures," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2018, pp. 11–20. DOI: `10.1109/SOSE.2018.00011`.

[3] Olaf Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3-4, pp. 301–310, 2017. DOI: `10.1007/s00450-016-0337-0`.

[4] P.A. Laplante, *What Every Engineer Should Know about Software Engineering*, ser. What Every Engineer Should Know. CRC Press, 2007, ISBN: 9781420006742.

[5] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, Manuel Mazzara and Bertrand Meyer, Eds. Cham: Springer International Publishing, 2017, pp. 195–216, ISBN: 978-3-319-67425-4. DOI: `10.1007/978-3-319-67425-4_12`.

[6] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," 2016. [Online]. Available: `https://arxiv.org/abs/1605.03175`.

[7] Claus Pahl and Pooyan Jamshidi, "Microservices: A systematic mapping study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, ser. CLOSER 2016, Rome, Italy: SCITEPRESS - Science and Technology Publications, Lda, 2016, pp. 137–146, ISBN: 9789897581823. DOI: `10.5220/0005785501370146`. [Online]. Available: `https://doi.org/10.5220/0005785501370146`.

[8] Wilhelm Hasselbring, “Microservices for scalability: Keynote talk abstract,” in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE ’16, Delft, The Netherlands: Association for Computing Machinery, 2016, pp. 133–134, ISBN: 9781450340809. DOI: `10.1145/2851553.2858659`. [Online]. Available: `https://doi.org/10.1145/2851553.2858659`.

[9] Yale Yu, H. Silveira, and M. Sundaram, “A microservice based reference architecture model in the context of enterprise architecture,” in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, 2016, pp. 1856–1860. DOI: `https://doi.org/10.1109/IMCEC.2016.7867539`.

[10] Sam Newman, *Building microservices: designing fine-grained systems*. " O’Reilly Media, Inc.", 2015.

[11] Mark Richards, “Microservices vs. service-oriented architecture,” 2015. [Online]. Available: `https://www.oreilly.com/radar/microservices-vs-service-oriented-architecture/`.

[12] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, “Microservices in practice, part 2: Service integration and sustainability,” *IEEE Software*, vol. 34, no. 02, pp. 97–104, Mar. 2017, ISSN: 1937-4194. DOI: `10.1109/MS.2017.56`.

[13] Washington Henrique Carvalho Almeida, Luciano de Aguiar Monteiro, Raphael Rodrigues Hazin, Anderson Cavalcanti de Lima, and Felipe Silva Ferraz, “Survey on microservice architecture-security, privacy and standardization on cloud computing environment,” *ICSEA 2017*, p. 210, 2017. [Online]. Available: `http://www.academia.edu/download/55044545/icsea_2017_full.pdf#page=211`.

[14] Christian Otterstad and Tetiana Yarygina, “Low-level exploitation mitigation by diverse microservices,” in *Service-Oriented and Cloud Computing*, Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen, Eds., Cham: Springer International Publishing, 2017, pp. 49–56, ISBN: 978-3-319-67262-5. DOI: `10.1007/978-3-319-67262-5_4`.

[15] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim, “Performance comparison analysis of linux container and virtual machine for building cloud,” *Advanced Science and Technology Letters*, vol. 66, no. 105-111, p. 2, 2014.

[16] Jim Smith and Ravi Nair, *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[17] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, Mar. 2007, ISSN: 0163-5980. DOI: `10.1145/1272998.1273025`.

[18] James Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[19] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[20] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski, "Scale-up x scale-out: A case study using nutch/lucene," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8. DOI: `10.1109/IPDPS.2007.370631`.

[21] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, "Supporting microservice evolution," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 539–543. DOI: `10.1109/ICSME.2017.63`.

[22] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina, "Microservices: How to make your application scale," in *Perspectives of System Informatics*, Alexander K. Petrenko and Andrei Voronkov, Eds., Cham: Springer International Publishing, 2018, pp. 95–104, ISBN: 978-3-319-74313-4.

[23] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *2019 IEEE International Conference on Web Services (ICWS)*, 2019, pp. 68–75. DOI: `10.1109/ICWS.2019.00023`.

[24] Andrew S. Tanenbaum and Maarten Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[25] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10. DOI: `10.1109/IISWC.2016.7581269`.

[26] Tetiana Yarygina, "Exploring microservice security," PhD thesis, The University of Bergen, Bergen, Norway, Oct. 2018.

[27] Fabrizio Montesi and Janine Weber, *Circuit breakers, discovery, and api gateways in microservices*, 2016. arXiv: `1609.05830 [cs.SE]`.

[28] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal, "Chaos engineering," *CoRR*, vol. abs/1702.05843, 2017. arXiv: `1702.05843`. [Online]. Available: `http://arxiv.org/abs/1702.05843`.

[29] Inna Vistbakka and Elena Troubitsyna, "Formalising privacy-preserving constraints in microservices architecture," in *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, Shang-Wei Lin, Zhe Hou, and Brendan Mahoney, Eds., ser. Lecture Notes in Computer Science, vol. 12531, Springer, 2020, pp. 308–317. DOI: `10.1007/978-3-030-63406-3\_19`.

[30] Howard B. Goodman and Pam Rowland, "Deficiencies of compliancy for data and storage - isolating the CIA triad components to identify gaps to security," in *National Cyber Summit (NCS) Research Track 2020, Huntsville, AL, USA, June 2-4, 2020*, Kim-Kwang Raymond Choo, Tommy Morris, Gilbert L. Peterson, and Eric Imsand, Eds., ser. Advances in Intelligent Systems and Computing, vol. 1271, Springer, 2020, pp. 170–192. DOI: `10.1007/978-3-030-58703-1\_11`.

[31] W. Diffie, "The first ten years of public-key cryptography," *Proceedings of the IEEE*, vol. 76, no. 5, pp. 560–577, 1988. DOI: `10.1109/5.4442`.

[32] E. Rescorla, "The transport layer security (tls) protocol version 1.3," RFC Editor, RFC 8446, Oct. 2018. DOI: `10.17487/RFC8446`. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc8446.txt`.

[33] Paul Lackner, "Anforderungen für den Aufbau eines eIDAS konformen Vertrauensdiensteanbieters," German, 2019. [Online]. Available: `https://phaidra.fhstp.ac.at/o:3624`.

[34] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile," RFC Editor, RFC 5280, Oct. 2018. DOI: `10.17487/RFC5280`. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc5280.txt`.

[35] Carl Ellison and Bruce Schneier, "Ten risks of pki: What you're not being told about public key infrastructure," *Comput Secur J*, vol. 16, no. 1, pp. 1–7, 2000. [Online]. Available: `https://cs.gmu.edu/~eoster/2019-795/2019-795-papers/paper-pki.pdf`.

[36] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich, "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, Samuel T. King, Ed., USENIX Association, 2013, pp. 399–314. [Online]. Available: `https:`

//www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/wang%5C_rui.

[37] E. Hammer-Lahav, "The OAuth 1.0 Protocol," RFC Editor, RFC 5849, 2010. DOI: 10.17487/RFC5849. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5849.txt.

[38] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC Editor, RFC 6749, 2012. DOI: 10.17487/RFC6749. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6749.txt.

[39] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "OpenID Connect Core 1.0," Tech. Rep., Nov. 2014. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html.

[40] Inna Vistbakka and Elena Troubitsyna, "Analysing privacy-preserving constraints in microservices architecture," in *44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020*, IEEE, 2020, pp. 1089–1090. DOI: 10.1109/COMPSAC48688.2020.0-126.

[41] C. Fetzer, "Building critical applications using microservices," *IEEE Security Privacy*, vol. 14, no. 6, pp. 86–89, 2016. DOI: 10.1109/MSP.2016.129.

[42] Emiliano Casalicchio and Stefano Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurr. Comput. Pract. Exp.*, vol. 32, no. 17, 2020. DOI: 10.1002/cpe.5668.

[43] Faheem Ullah, Adam Johannes Raft, Mojtaba Shahin, Mansooreh Zahedi, and Muhammad Ali Babar, "Security support in continuous deployment pipeline," in *ENASE 2017 - Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, Porto, Portugal, April 28-29, 2017*, Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek, Eds., SciTePress, 2017, pp. 57–68. DOI: 10.5220/0006318200570068. [Online]. Available: https://doi.org/10.5220/0006318200570068.

[44] M. Pahl, F. Aubet, and S. Liebald, "Graph-based iot microservice security," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–3. DOI: 10.1109/NOMS.2018.8406118.

[45] Y. Sun, S. Nanda, and T. Jaeger, "Security-as-a-service for microservices-based cloud applications," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 50–57. DOI: `10.1109/CloudCom.2015.93`.

[46] Dalton A. Hahn, Drew Davidson, and Alexandru G. Bardas, "Security issues and challenges in service meshes - an extended study," *CoRR*, vol. abs/2010.11079, 2020. arXiv: `2010.11079`. [Online]. Available: `https://arxiv.org/abs/2010.11079`.

[47] Jorge Werner, Carla Merkle Westphall, and Carlos Becker Westphall, "Cloud identity management: A survey on privacy strategies," *Computer Networks*, vol. 122, pp. 29–42, 2017. DOI: `10.1016/j.comnet.2017.04.030`.

[48] Issa M Khalil, Abdallah Khreishah, and Muhammad Azeem, "Cloud computing security: A survey," *Computers*, vol. 3, no. 1, pp. 1–35, 2014. DOI: `10.3390/computers3010001`.

[49] C. Saravanakumar and C. Arun, "Survey on interoperability, security, trust, privacy standardization of cloud computing," in *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, 2014, pp. 977–982. DOI: `10.1109/IC3I.2014.7019735`.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg, "Meltdown," *CoRR*, vol. abs/1801.01207, 2018. arXiv: `1801.01207`. [Online]. Available: `http://arxiv.org/abs/1801.01207`.

[51] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19. DOI: `10.1109/SP.2019.00002`.

[52] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 753–768, ISBN: 9781450367479. DOI: `10.1145/3319535.3354252`. [Online]. Available: `https://doi.org/10.1145/3319535.3354252`.

[53] Yuval Yarom and Katrina Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732, ISBN: 978-1-931971-15-7. [Online]. Available:

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[54] Vasilios Mavroudis, Andrea Cerulli, Petr Svenda, Dan Cvrcek, Dusan Klinec, and George Danezis, "A touch of evil: High-assurance cryptographic hardware from untrusted components," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1583–1600, ISBN: 9781450349468. DOI: 10.1145/3133956.3133961. [Online]. Available: https://doi.org/10.1145/3133956.3133961.

[55] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016. DOI: 10.1109/MCC.2016.100.

[56] Michael Pearce, Sherali Zeadally, and Ray Hunt, "Virtualization: Issues, security threats, and solutions," *ACM Comput. Surv.*, vol. 45, no. 2, Mar. 2013, ISSN: 0360-0300. DOI: 10.1145/2431211.2431216. [Online]. Available: https://doi.org/10.1145/2431211.2431216.

[57] B. R. Kandukuri, R. P. V., and A. Rakshit, "Cloud security issues," in *2009 IEEE International Conference on Services Computing*, 2009, pp. 517–520. DOI: 10.1109/SCC.2009.84.

[58] R. Sravan Kumar and A. Saxena, "Data integrity proofs in cloud storage," in *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*, 2011, pp. 1–4. DOI: 10.1109/COMSNETS.2011.5716422.

[59] S. Abdullah and K. A. Abu Bakar, "Security and privacy challenges in cloud computing," in *2018 Cyber Resilience Conference (CRC)*, 2018, pp. 1–3. DOI: 10.1109/CR.2018.8626872.

[60] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann, "Restful web services vs. "big"' web services: Making the right architectural decision," in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW '08, Beijing, China: Association for Computing Machinery, 2008, pp. 805–814, ISBN: 9781605580852. DOI: 10.1145/1367497.1367606.

[61] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14, Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488, ISBN: 9781450332132. DOI: 10.1145/2663716.2663755. [Online]. Available: https://jhalderm.com/pub/papers/heartbleed-imc14.pdf.

[62] Bodo Möller, Thai Duong, and Krzysztof Kotowicz, “This POODLE Bites: Exploiting The SSL 3.0 Fallback,” Sep. 2014. [Online]. Available: `https://www.openssl.org/~bodo/ssl-poodle.pdf`.

[63] Muhammad Baqer Mollah, Md Abul Kalam Azad, and Athanasios Vasilakos, “Security and privacy challenges in mobile cloud computing: Survey and way ahead,” *Journal of Network and Computer Applications*, vol. 84, pp. 38–54, 2017. DOI: `10.1016/j.jnca.2017.02.001`.

[64] Chris Richardson and Floyd Smith, *Microservices: From Design to Deployment*. nginx. [Online]. Available: `https://www.nginx.com/resources/library/designing-deploying-microservices/`.

[65] Paul Lackner., “How to mock a bear: Honeypot, honeynet, honeywall & honeytoken: A survey,” in *Proceedings of the 23rd International Conference on Enterprise Information Systems - Volume 2: ICEIS,*, INSTICC, 2021, ISBN: 978-989-758-509-8. DOI: `10.5220/0010400001810188`.

[66] R. Roostaei and Z. Movahedi, “Mobility and context-aware offloading in mobile cloud computing,” in *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, 2016, pp. 1144–1148. DOI: `10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0176`.

[67] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas, “Towards the understanding and evolution of monolithic applications as microservices,” in *2016 XLII Latin American Computing Conference (CLEI)*, 2016, pp. 1–11. DOI: `10.1109/CLEI.2016.7833410`.

[68] D. Jaramillo, D. V. Nguyen, and R. Smart, “Leveraging microservices architecture by using docker technology,” in *SoutheastCon 2016*, 2016, pp. 1–5. DOI: `10.1109/SECON.2016.7506647`.

[69] Madiha H. Syed and Eduardo B. Fernández, “A reference architecture for the container ecosystem,” in *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018*, Sebastian Doerr, Mathias Fischer, Sebastian Schrittwieser, and Dominik Herrmann, Eds., ACM, 2018, 31:1–31:6. DOI: `10.1145/3230833.3232854`. [Online]. Available: `https://doi.org/10.1145/3230833.3232854`.

[70] Zhiqiang Jian and Long Chen, “A defense method against docker escape attack,” in *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy, ICCSP 2017, Wuhan, China, March 17 - 19, 2017*, ACM, 2017, pp. 142–146. DOI: `10.1145/3058060.3058085`. [Online]. Available: `https://doi.org/10.1145/3058060.3058085`.

[71] Yang Luo, Wu Luo, Xiaoning Sun, Qingni Shen, Anbang Ruan, and Zhonghai Wu, "Whispers between the containers: High-capacity covert channel attacks in docker," in *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, IEEE, 2016, pp. 630–637. DOI: `10.1109/TrustCom.2016.0119`. [Online]. Available: `https://doi.org/10.1109/TrustCom.2016.0119`.

[72] Amith Raj MP, A. Kumar, S. J. Pai, and A. Gopal, "Enhancing security of docker using linux hardening techniques," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2016, pp. 94–99. DOI: `10.1109/ICATCCT.2016.7911971`.

[73] Reid Priedhorsky and Tim Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in HPC," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, Bernd Mohr and Padma Raghavan, Eds., ACM, 2017, 36:1–36:10. DOI: `10.1145/3126908.3126925`. [Online]. Available: `https://doi.org/10.1145/3126908.3126925`.

[74] Abdulrahman Azab, "Enabling docker containers for high-performance and many-task computing," in *2017 IEEE International Conference on Cloud Engineering, IC2E 2017, Vancouver, BC, Canada, April 4-7, 2017*, IEEE Computer Society, 2017, pp. 279–285. DOI: `10.1109/IC2E.2017.52`. [Online]. Available: `https://doi.org/10.1109/IC2E.2017.52`.

[75] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer, "SCONE: secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe, Eds., USENIX Association, 2016, pp. 689–703. [Online]. Available: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov`.

[76] Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni, "Securestreams: A reactive middleware framework for secure data stream processing," *CoRR*, vol. abs/1805.01752, 2018. arXiv: `1805.01752`. [Online]. Available: `http://arxiv.org/abs/1805.01752`.

[77] Casen Hunger, Lluıs Vilanova, Charalampos Papamanthou, Yoav Etsion, and Mohit Tiwari, "DATS - data containers for web applications," in *Proceedings of the Twenty-Third International Confer-*

ence on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, Eds., ACM, 2018, pp. 722–736. DOI: `10.1145/3173162.3173213`. [Online]. Available: `https://doi.org/10.1145/3173162.3173213`.

[78] Ioannis Giannakopoulos, Konstantinos Papazafeiropoulos, Katerina Doka, and Nectarios Koziris, "Isolation in docker through layer encryption," in *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, Kisung Lee and Ling Liu, Eds., IEEE Computer Society, 2017, pp. 2529–2532. DOI: `10.1109/ICDCS.2017.161`. [Online]. Available: `https://doi.org/10.1109/ICDCS.2017.161`.

[79] Alireza Ranjbar, Miika Komu, Patrik Salmela, and Tuomas Aura, "Synaptic: Secure and persistent connectivity for containers," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*, IEEE, 2017, pp. 262–267. DOI: `10.1109/CCGRID.2017.62`. [Online]. Available: `https://doi.org/10.1109/CCGRID.2017.62`.

[80] Xin Sun, Yu-Wei Eric Sung, Sunil Krothapalli, and Sanjay G. Rao, "A systematic approach for evolving VLAN designs," in *INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March 2010, San Diego, CA, USA*, IEEE, 2010, pp. 1451–1459. DOI: `10.1109/INFCOM.2010.5461961`. [Online]. Available: `https://doi.org/10.1109/INFCOM.2010.5461961`.

[81] Keith Kirkpatrick, "Software-defined networking," *Commun. ACM*, vol. 56, no. 9, pp. 16–19, 2013. DOI: `10.1145/2500468.2500473`. [Online]. Available: `https://doi.org/10.1145/2500468.2500473`.

[82] Florian Malecki, "Next-generation firewalls: Security with performance," *Netw. Secur.*, vol. 2012, no. 12, pp. 19–20, 2012. DOI: `10.1016/S1353-4858(12)70114-9`. [Online]. Available: `https://doi.org/10.1016/S1353-4858(12)70114-9`.

[83] Shiyao Ma, Jingjie Jiang, Bo Li, and Baochun Li, "Maximizing container-based network isolation in parallel computing clusters," in *24th IEEE International Conference on Network Protocols, ICNP 2016, Singapore, November 8-11, 2016*, IEEE Computer Society, 2016, pp. 1–10. DOI: `10.1109/ICNP.2016.7784434`. [Online]. Available: `https://doi.org/10.1109/ICNP.2016.7784434`.

[84] Amr Osman, Armin Wasicek, Stefan Köpsell, and Thorsten Strufe, "Transparent microsegmentation in smart home iot networks," in *3rd USENIX Workshop on Hot Topics in Edge Computing, HotEdge 2020, June 25-26, 2020*, Irfan Ahmad and Ming Zhao, Eds., USENIX Association, 2020. [Online]. Available: `https://www.usenix.org/conference/hotedge20/presentation/osman`.

[85] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, Eds., ACM, 2017, pp. 197–207. DOI: `10.1145/3106237.3106270`. [Online]. Available: `https://doi.org/10.1145/3106237.3106270`.

[86] Volker Gruhn, Christoph Hannebauer, and Christian John, "Security of public continuous integration services," in *Proceedings of the 9th International Symposium on Open Collaboration, Hong Kong, China, August 05 - 07, 2013*, Ademar Aguiar and Dirk Riehle, Eds., ACM, 2013, 15:1–15:10. DOI: `10.1145/2491055.2491070`. [Online]. Available: `https://doi.org/10.1145/2491055.2491070`.

[87] Ken Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984. DOI: `10.1145/358198.358210`. [Online]. Available: `https://doi.org/10.1145/358198.358210`.

[88] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly, "Zero trust architecture," NIST, Tech. Rep., Aug. 2020. DOI: `10.6028/NIST.SP.800-207`.

[89] Tetiana Yarygina, "Restful is not secure," in *Applications and Techniques in Information Security*, Lynn Batten, Dong Seong Kim, Xuyun Zhang, and Gang Li, Eds., Singapore: Springer Singapore, 2017, pp. 141–153, ISBN: 978-981-10-5421-1.

[90] Roy T Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000, vol. 7.

[91] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590. DOI: `10.1109/ColumbianCC.2015.7333476`.

[92] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a smart city internet of things platform with microservice architecture," in *2015 3rd International Conference on Future Internet of Things and Cloud*, 2015, pp. 25–30. DOI: `10.1109/FiCloud.2015.55`.

[93] Cristian Gadea, Mircea Trifan, Dan Ionescu, and Bogdan Ionescu, "A reference architecture for real-time microservice api consumption," in *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, ser. CrossCloud '16, London, United Kingdom: Association for Computing Machinery, 2016, ISBN: 9781450342940. DOI: `10.1145/2904111.2904115`. [Online]. Available: `https://doi.org/10.1145/2904111.2904115`.

[94] Steve McConnell, *Code Complete: A Practical Handbook of Software Construction*. Microsoft, ISBN: 978-0735619678.

[95] Russ Cox, "Surviving software dependencies," *Commun. ACM*, vol. 62, no. 9, pp. 36–43, 2019. DOI: `10.1145/3347446`. [Online]. Available: `https://doi.org/10.1145/3347446`.

[96] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *3rd USENIX Workshop on Hot Topics in Edge Computing, HotEdge 2020, June 25-26, 2020*, Irfan Ahmad and Ming Zhao, Eds., USENIX Association, 2020. [Online]. Available: `https://www.usenix.org/conference/hotedge20/presentation/fu`.

[97] Alexandre Decan, Tom Mens, and Maelick Claes, "On the topology of package dependency networks: A comparison of three programming language ecosystems," in *Proccedings of the 10th European Conference on Software Architecture Workshops*, ser. ECSAW '16, Copenhagen, Denmark: Association for Computing Machinery, 2016, ISBN: 9781450347815. DOI: `10.1145/2993412.3003382`. [Online]. Available: `https://doi.org/10.1145/2993412.3003382`.

[98] Bo Wang, Hengrui Ma, Xunting Wang, Guiping Deng, Yan Yang, and Shaohua Wan, "Vulnerability assessment method for cyber-physical system considering node heterogeneity," *The Journal of Supercomputing*, pp. 1–21, 2019. DOI: `10.1007/s11227-019-03027-w`.

[99] Bryan Payne, "PKI at scale using short–lived certificates," San Francisco, CA: USENIX Association, Jan. 2016. [Online]. Available: `https://www.usenix.org/conference/enigma2016/conference-program/presentation/payne`.

[100] Ronald L. Rivest, "Can we eliminate certificate revocation lists?" In *Financial Cryptography*, Rafael Hirchfeld, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 178–183, ISBN: 978-3-540-53918-6.

[101] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster, "The dos and don'ts of client authentication on the web.," in *USENIX Security Symposium*, 2001, pp. 251–268.

[102] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 933–940, 1987. DOI: `10.1109/TC.1987.5009516`.

[103] Ronald L. Rivest and Butler Lampson, "SDSI-a simple distributed security infrastructure," Crypto, 1996.

[104] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen, *RFC2693: SPKI Certificate theory*, 1999.

[105] "Profiles for the OASIS SecurityAssertion Markup Language (SAML)V2.0," OASIS, 2005. [Online]. Available: `https://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf`.

[106] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser, "N-variant systems: A secretless framework for security through diversity.," in *USENIX Security Symposium*, 2006, pp. 105–120. [Online]. Available: `https://www.usenix.org/legacy/event/sec06/tech/full_papers/cox/cox.pdf`.

[107] Tetiana Yarygina and Christian Otterstad, "A game of microservices: Automated intrusion response," in *IFIP International Conference on Distributed Applications and Interoperable Systems*, Springer, 2018, pp. 169–177. DOI: `10.1007/978-3-319-93767-0_12`.

[108] Gianluigi Folino and Pietro Sabatino, "Ensemble based collaborative and distributed intrusion detection systems: A survey," *Journal of Network and Computer Applications*, vol. 66, pp. 1–16, 2016. DOI: `10.1016/j.jnca.2016.03.011`.

[109] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz, "Compiler-generated software diversity," in *Moving Target Defense*, Springer, 2011, pp. 77–98. DOI: `10.1007/978-1-4614-0977-9_4`.

[110] Mohammad A Noureddine, Ahmed Fawaz, William H Sanders, and Tamer Başar, "A game-theoretic approach to respond to attacker lateral movement," in *International Conference on Decision and*

Game Theory for Security, Springer, 2016, pp. 294–313. DOI: 10.1007/978-3-319-47413-7_17.

[111] A. Avizienis, J. -. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. DOI: 10.1109/TDSC.2004.2.

[112] James Lewis and Martin Fowler, *Microservices*, Mar. 25, 2014. [Online]. Available: https://www.martinfowler.com/articles/microservices.html.

[113] C. Perra, "A framework for the development of sustainable urban mobility applications," in *2016 24th Telecommunications Forum (TELFOR)*, 2016, pp. 1–4. DOI: 10.1109/TELFOR.2016.7818788.

[114] Kaibin Bao, Ingo Mauser, Sebastian Kochanneck, Huiwen Xu, and Hartmut Schmeck, "A microservice architecture for the intranet of things and energy in smart buildings: Research paper," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, ser. MOTA '16, Trento, Italy: Association for Computing Machinery, 2016, ISBN: 9781450346696. DOI: 10.1145/3007203.3007215. [Online]. Available: https://doi.org/10.1145/3007203.3007215.

[115] S. Srinivasan, "Data privacy concerns involving cloud," in *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2016, pp. 53–56. DOI: 10.1109/ICITST.2016.7856665.

[116] Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika Mäntylä, and Tomi Männistö, "The highways and country roads to continuous deployment," *IEEE Softw.*, vol. 32, no. 2, pp. 64–72, 2015. DOI: 10.1109/MS.2015.50.