KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle 30 credits

Evaluation of security threats in microservice architectures

WILLIAM LINDBLOM

Stockholm, Sweden

Evaluation of security threats in microservice architectures

WILLIAM LINDBLOM

Master's Programme, Computer Science, 120 credits
Date: August 14, 2022

Supervisor: Giuseppe Nebbione
Examiner: Mathias Ekstedt
School of Electrical Engineering and Computer Science
Host company: OmegaPoint AB
Swedish title: Evaluering av säkerhetshot i mikrotjänst arkitekturer

Abstract

The microservice architecture is a popular architectural pattern in the industry to implement large systems as they can reduce the code bases of each service and increase the maintainability for each of the individual services by dividing the application into smaller components based on business logic. The services can be implemented in different programming languages and communicates over a network. As a consequence, it might lead to a greater attack surface for an adversary of the system. In order to ease the implementation of microservice architectures, a set of design patterns exists. Two patterns addressing the security of the architecture are the API Gateway pattern and the sidecar pattern. More research is needed in order to identify the security threats microservice architecture encounters and how the design pattern handles those. This master thesis uses threat modeling with attack graphs along with attack simulations in order to investigate the threats in microservice architectures and how they compare between the design patterns. To construct the attack graphs and perform the attack simulations SecuriCAD along with CoreLang was used on a microservice architecture with each of the design patterns. The report concludes that the sidecar pattern is faced with less risk than the API Gateway pattern overall and presents a set of suggestions regarding how the security can be improved in microservice architectures.

Keywords

Microservice architectures, Threat modelling, Design patterns

Sammanfattning

Mikrotjänstarkitekturer har blivit ett populärt arkitekturmönster inom industrin för att implementera större system eftersom det kan reducera kodbaserna och underlätta underhållningen av varje enskild tjänst genom av att dela upp applikationen i mindre komponenter baserat på varje tjänsts domänlogik. Dessa tjänster kan vara implementerade i olika programmeringsspråk och kommunicerar med varandra över ett nätverk. Som följd skulle dock detta kunna leda till en större attackyta för en angripare av systemet. För att underlätta implementationen av mikrotjänster finns en mängd designmönster, två designmönster som hanterar säkerheten av mikrotjänstarkiterurer är API Gateway mönstret och sidecar mönstret. Mer forskning skulle dock behövas för att ta reda på vilka hot som mikrotjänstarkitekturer ställs inför samt hur väl de två design mönstren bemöter dessa. Den här masteruppsatsen använder hotmodellering med attack grafer samt attack simuleringar för att undersöka vilka hot som finns i mikrotjänstarkitekturer och hur dessa skiljer sig åt mellan de två design mönstren. För att framställa attack graferna och genomföra attack simuleringarna användes programmet SecuriCAD tillsammans med CoreLang på en mikrotjänstarkitektur med vardera design mönster. Rapporten kommer fram till att sidecarmönstrer har lägre risk i jämförelse med API Gateway mönstret överlag och presenterar en mängd förslag angående hur säkerheten kan förbättras i mikrotjänstarkitekturer.

Nyckelord

Mikrotjänstarkitekturer, Hotmodellering, Designmönster

Acknowledgments

Finalizing this thesis would not have been possible without the assistance of all the people involved in the project. For guidance, discussing ideas, and feedback throughout the writing of this thesis I would like to thank my supervisor at Omegapoint, Daniel Muresu. For making this thesis work possible and allowing me to work on the project, I would like to thank the hosting company Omegapoint. For fulfilling the role as academic supervisor, providing feedback on the report, guidance in SecuriCAD, and valuable insights into threat modeling I would like to thank Giuseppe Nebbione. For iterations of the thesis work in the initial phase, connecting me with Giuseppe and for taking the role of examiner for this project, I would like to thank Mathias Ekstedt.

Stockholm, August 2022
William Lindblom

Contents

List of Figures

List of Tables

Acronyms

MSA Microservice Architecture

REST Representational State Transfer

RPC Remote Procedure Call

Chapter 1

Introduction

The chapter aims to provide an overview of what this thesis will investigate. First, a background of the subject matter will be given describing microservice architectures and threat modeling. In section 1.2 the problem and research questions will be provided. In section 1.3 the purpose of the thesis will be presented along with its ethical and sustainability impacts. Then the goals along with a brief description of the research methodology will be given in section 1.4 and 1.5 respectively. In section 1.6 the scope of the thesis along with its delimitations will be given and lastly, in section 1.7 the outline of the thesis will be given.

1.1 Background

This thesis will evaluate the security implications of utilizing a microservice architecture (MSA) as the architectural pattern by investigating how two common security design patterns handle threats to the system.

The transformation from traditional monolithic architectures toward MSAs has been an increasing trend in the industry in recent years when it comes to implementing computer systems. The MSAs is an architectural pattern to make it possible to develop scalable and maintainable IT systems where the systems are divided into minimalistic services based on business logic. By dividing the system into small independent components the size of each service code base can be reduced, which can have a positive impact on the maintainability of the system. The services communicate with each other via messages commonly by utilizing REST or remote procedure calls (RPC) and are often deployed containerized in a distributed system to improve scalability.

As MSAs rely heavily on network communication to interact with each

other security problems related to the network layer become more prominent compared to a monolithic application. Furthermore, as the microservices are often containerized, problems related to container security can also be present in a MSA. The individual services in the architecture have a similar structure as web servers, it is therefore possible that the security challenges present for web servers are also present in MSAs.

Addressing the security of a system can be a challenging task as it requires a deep understanding of the system under evaluation. A widely used method in order to address security-related problems within a system is by utilizing threat modeling. This thesis will use the threat modeling technique of attack graphs along with attack simulations in order to identify the weaknesses of the system.

1.2 Problem

Migrating from a monolithic architecture to a MSA can be beneficial for scalability and maintainability. Since the MSA has more services it is naturally less centralized compared to a monolith, furthermore, the services should operate independently of machine architecture and programming language. On the other hand, in the case of a MSA the system is divided into minimalistic components which can lead to a larger attack surface for a malicious actor. To address security-related problems in MSAs there exists design patterns such as the API Gateway pattern and the sidecar pattern, but further research is needed to gain insights into how the design patterns retain the authenticity, confidentiality, integrity, and availability of the systems. This thesis aims to answer the question:

> ***RQ1:*** *What threats are present in microservice architectures?*
>
> ***RQ2:*** *How do the risks compare between the API Gateway pattern and the sidecar pattern?*

1.3 Purpose

The purpose of this thesis is to evaluate the security aspects related to microservices along with the security implications of utilizing specific design patterns. Microservices can be implemented in a variety of programming languages and technologies with their own set of security implications. This thesis will take a more abstract approach where the objective is to identify

characteristics of two common microservice design patterns with respect to security. Thus the thesis will focus on potential attacks and critical paths for an adversary with the aim to compromise the MSA regardless of the underlying technologies.

1.3.1 Ethics

The project will not investigate or evaluate any microservices in production, hence no users or humans will be directly affected by the result. However like any security research, pointing out potential vulnerabilities in computer systems might be used by a malicious actor in a way not intended by this project.

1.3.2 Sustainability

The project contributes to sustainability by providing knowledge about possible threats in computer systems and thus has the potential to help organizations mitigate the risks of becoming victims of computer crimes.

1.4 Goals

The goal of this thesis is to provide knowledge regarding the security implications for MSAs to the hosting company, Omegapoint, as well as to the industry. This will be accomplished by performing threat modeling and attack simulations to identify the risks of the two design patterns commonly used in MSAs in order to evaluate the consequences of potential attack scenarios. The aim of this project is to deliver suggestions regarding the actions that can be taken to increase the security both for the MSA as a whole but also suggested mitigation's concerning the specific design patterns.

1.5 Research Methodology

In order to answer the research questions, two problems need to be solved. The first problem is to identify the threats in the different MSAs. The second problem is to calculate the risks for the respective design patterns. To address the aforementioned problems a quantitative research method will be used. The first problem will be addressed by generating a threat model for the respective MSAs. The second problem will be addressed by performing attack

simulations on the MSAs to retrieve a quantifiable result for the threats in terms of critical paths and the time it takes for an adversary to compromise the system.

In order to validate the results, the project will utilize a qualitative method to compare if threats identified in previous research correlate to the threats found in this thesis. So forth the results will not solely depend on attack simulations but also be backed by qualitative methods.

1.6 Delimitations

The MSA is a generic architectural pattern and is not bound to the implementation of any specific technology or underlying hardware. Therefore this project will not investigate potential attacks specific to particular technologies, programming languages, or hardware-related problems.

There exist several different design patterns for MSAs. This project will only focus on the design patterns given in the research question. Furthermore, finding threats in a system is a large task, this project will use threat modeling with the use of probabilistic attack graphs and thus potentially not all attack scenarios will be discovered.

1.7 Structure of the thesis

Chapter 3 presents the background information about microservice architectures along with the design patterns. Chapter 4 presents the background information regarding threat modeling and attack simulations. In Chapter 5 the previous background chapters are connected with a presentation of identified threats to microservice architectures along with related work in the field. Chapter 6 describes the methods used in the project and Chapter 7 presents the implemented asset models of the design patterns. In Chapter 8 the results of the attack simulations are presented and analyzed along with suggestions regarding how the security can be improved within the microservice architectures. Finally, in Chapter 9 the conclusions of the thesis along with the limitations and future work are presented.

Chapter 2

Related work area

Related to this thesis are previous work about threats specific to a MSA and previous work about conducting threat modeling of MSAs. Section 2.3.1 presents related work of threats to MSAs and Section 2.3.2 presents related work of threat modeling in MSAs.

2.1 Microservice architectures

Related to microservice architectures is the work by Miller et al. [1]. The article addresses data leaks and breaches in MSAs by providing a proof of concept infrastructure based on the zero-trust principles. The aim of the article is to enable data to be exchanged by agents not trusting each other where the data is secure both at rest and in transit.

According to the article, the likelihood of data exposure has increased because public clouds are increasingly used by companies to process data and because the data is moved frequently. As a consequence this leads to the losses both in terms of user privacy and in money for companies. The article further mentions that the data exposure has been a consequence of both malicious actors and also of accidents for example as misconfigurations of endpoints or databases. It is further highlighted that as the attack surface of MSAs is increased it is not enough to secure the network boundaries only. To mitigate theses problems the article proposes to use the zero-trust security model, that is, to authenticate and authorize all of the traffic flows in the system.

In order to achieve this the article suggests an infrastructure based on pods consisting of three containers, one container for the actual service, one container as a sidecar proxy and one container as a sidecar enforcing policies. That is, the sidecar proxy is responsible for interception all traffic and the

policy sidecar is responsible for whether or not the traffic should be authorized. In order to secure the data at rest the article proposes to store the data encrypted on mounted Persistent Volumes (PVs) and in order to secure the data in transport the article proposes to use mTLS for the communication between the pods. On the other hand, the article argues that the internal communication within the pod does not need to be encrypted because it is already protected by the isolation layers.

The article further benchmark the implementation to investigate the overhead of the infrastructure. This part is not particularly relevant for this thesis but it is worth mentioning that they conclude that the overhead regarding start up time is two seconds on average and either 7% or 65% overhead in the request time depending on what level of order the policy size is set to.

2.2 Threat modeling

Related to the area of threat modeling is the work by Shevchenko et. al. [2]. The contribution of the article is that it provides a summary of available methods for threat modeling along with a comparison of the methods and therefore provides an overview of existing methods for threat modeling and previous work in the field. The article explains that software systems are opposed to both internal and external threats and that threat modeling can be used in order to understand how to defend the system. With the use of threat modeling in the beginning of the systems the article highlights that a proactive approach can be taken in order to make architectural decisions regarding the implementation of the system.

The article discusses the twelve different threat modeling methods STRIDE, PASTA, LINDDUN, CVSS, Attack Trees, Persona non Grata, Security Cards, hTMM, Quantitative Threat Modeling Method, Trike, VAST Modeling and OCTAVE. While no detailed description will be given for each of the threat modeling approaches, the article concludes that STRIDE is the most mature method and both STRIDE and attack graphs are suitable in order to identify mitigations. The article further concludes that attack graphs are easy to understand if the user have previous knowledge of the system and that the results are consistent.

Related to the area of automatic attack graph generation is, in addition to MAL, the work of P. Johnson et al. with the work of pwnPr3d [3]. The contribution of the paper is the threat modeling technique pwnPr3d, a attacker-centric modeling language which provides the functionality to model assets, the attacker and the attack steps. Furthermore it provides the functionality to

model relations between assets along with probability distributions, in order to be able to quantify the time to compromise. Left as future work in the paper is to extend the work of pwnPr3d in order to provide the functionality for users to instantiate their components. This problem was later addressed by MAL.

2.3 Microservice architecture threat modeling

In this section the previous work regarding threat modeling in MSAs is presented. The previous work can be divided in to two categories, the work of identifying threats and risks in MSA and the work of applying the threat modeling approach on MSAs. Section 2.3.1 presents previous work identify threats and risks in MSAs and Section 2.3.2 presents previous work of how threat modeling can be applied to MSAs.

2.3.1 Security in Microservice architectures

Related to threats and risks of MSAs is the work by Dongjin et al. [4]. The article is a survey with the goal to present risks and threats in the communication of microservices with respect to, containers, data, permission, and network.

In the article it is explained that containers face two main adversaries, direct- and indirect adversaries. The goal of direct adversaries is to modify or destroy system or network files, hence the target for direct adversaries are core services in the container. Indirect adversaries on the other hand targets the image repositories and code in the ecosystem of the container. The article presents several container related threats including Kernel exploit, Denial of Service, Container escapes, Poisoned images and Secret compromise. The threat of kernel exploit is present if the host operating system runs multiple container instances. An attacker would then try to gain unauthorized access to the containers without the user knowing. This is closely related to container escape attacks where the attacker is exploiting a kernel vulnerability to break out of the container. Once the attacker has managed to break out of the container a shell can be launched and the malicious actor can start to attack other containers. Furthermore the article explains the importance of resource isolation as it claims Denial of Service is a common attack in multi-tenant systems where the system is shared. To mitigate this the article proposes the use of cgroups, a type of container resource isolation to provide control

over the share of processes, groups, memory and CPU. The threat of image poisoning brings further consequences in a multi-tenant systems as the tenants share network address. If a malicious program runs hidden in a containers image, it will affect the reputation of all other tenants sharing the same resource group due to the share network address. Hence legitimate tenants might be denied access to resources because of the poisoned image.

Regarding data the article claims that interception and inference of the message data poses a threat to the MSA. Furthermore it explains how leaks of data in transfer is a challenge. In this part the article mainly focus on encryption and proposes that a combination of symmetric and asymmetric encryption is recommended for data in transfer. Furthermore the article presents other issues to microservices such as data tampering and destruction of the database.

The issues related to permissions originates from that microservices in general runs in a distributed cloud environment. Consequently the permission related security issues originates from ineffective access control with potential attacks including identity spoofing, illegal access, password leaks and replay attacks. The article describes the importance of verifying the authenticity of each service in the MSA as if a single service is compromised, it may try to alter other services in a malicious manner.

For network related security issues the article addresses both traditional network threats such as Man-in-the-middle (MITM) attacks, Denial of Service (DoS) and Address resolution (ARP) spoofing. The article also highlights that a MSA in nature requires communication among more services compared to a monolithic architecture. Consequently more endpoints are exposed leading to a greater attack surface toward network based attacks. Other network threats relates to the infrastructure the MSA utilizes. To increase efficiency and flexibility, Software Defined Networks (SDN) can be used. SDNs might pose threats to authentication and authorization for the network applications.

2.3.2 Microservice Architectures Threat Modeling

Related to threat modeling of MSAs is the work by Nkomo et al. [5]. The goal of the article is to identify security threats as consequences from flaws related to the design of the microservice compositions. The article proposes the following five security challenges related to MSAs: Increased surface attack, Indefinable security perimeters, Security monitoring is complex, Authentication is centralized and Threat modeling and risk assessment is localized. This is not relevant to this project per se, but the challenges provide

insights that facilitate the problems in this work. To address the challenges, the article suggests that an architecture-centric approach is most suitable for MSAs as it allows to identify threats in each component of the architecture. The architecture-centric approach used by the article is the Microsoft threat modeling process which consists of five steps:

1. Identify microservices composition security objectives.
2. Microservices composition overview.
3. Dissection of a Microservices composition.
4. Identifying threats to Microservices composition.
5. Identifying vulnerabilities of a Microservices composition.

The steps are related for identifying threats to MSAs in general and the second step relates to the essential components of the architecture which is related to the microservice design patterns in particular.

In step three and four threats specific to each microservices composition is identified. The compositions the article argues as potential entry points for an attacker are the API Gateway and the microservices API, the service registery, The message broker and the container or virtual machine running the microservice. However these compositions depends on the microservice design patterns even though the specific compositions are common in any MSA. Furthermore the article concludes that the threats related to the entry points are insecure application programming interfaces, unauthorized access, insecure microservice discovery, insecure runtime infrastructure and insecure message broker.

In step five, the threats for each component in the architecture is analyzed. To accomplish this the article utilizes the STRIDE framework to identify potential vulnerabilities with respect to spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege.

Chapter 3

Microservice Architectures

In order to understand what problems the MSA aims to solve, it is necessary to first explain the monolithic architecture. In [6] it is explained that the traditional architectural style of developing enterprise applications is by utilizing the monolithic architecture. In practice, this often means that the application consists of three parts. A user interface where the user interacts with the application on the client side. The server-side logic of the application to handle HTTP requests and responses along with the application logic and at last, the database to store, write and read data used by the application. Regarding the downsides of the monolithic architecture, they highlight two types of problems arising from containing all server-side logic in one component. The first problem is that as the complexity of the application grows, it becomes harder to update features as the whole application needs to be rebuilt and the second problem is that as it contains one server-side component, the whole application needs to be scaled upon increased load even if only a few parts of the application is concerned by the requests. In order to address the aforementioned problems, the MSA can be used.

In a MSA the application is divided into a set of microservices. The microservices are explained in [7] as a set of independent services designed to be loosely coupled with high cohesion communicating via messages over a network. They emphasize that the size of each service is supposed to be small and provide a single business capability in order to preserve granularity. Consequently, a MSA can be beneficial for maintainability and scalability which has led to increased interest from the industry.

However, there are natural trade-offs followed by the use of the MSA. As a MSA is a distributed system, security-related problems become more prominent. One such problem is that as more services are present, independent

of machine architectures and programming languages, a larger attack surface is exposed to malicious actors [7]. Another security-related problem is that according to OWASP, in the design phase of a microservice, authentication and authorization is one fundamental security requirement that needs to be addressed [8]. Furthermore, if the MSA includes third-party services, additional authentication challenges towards these services arise [7].

Naturally to microservices, it is described in [9] that sophisticated communication mechanisms are needed for communication between the services in comparison to a monolithic architecture. This results in a set of core features needed by every MSA such as access control, authentication, service discovery, response caching, load balancing, health checks, and monitoring. They also emphasize that depending on the design pattern, these features can either be bundled together or implemented as independent features. Furthermore, the features can either be implemented directly in the code base or as a part of the service composition.

3.1 Container-based virtualization

Container-based virtualization is a technique that provides an abstraction layer for the access paths between different software resources on an operating system by isolating a process with its own root file system, process tree, and network subsystem [10]. It should be noted that the technique is not unique to MSAs alone but commonly utilized in order to deploy new services. Containers in MSAs are further elaborated on in [11] describing that isolating the processes on the system level is different from hypervisor-based isolation which provides an abstraction layer from the hardware. Thus the processes in container-based virtualization share the operating system along with the system resources with other running processes. One container implementation is Docker which enables sharing of system libraries in addition to the shared operating system, thus making it more lightweight and scalable compared to hypervisor-based techniques such as virtual machines, which is why they mean Docker containers are more widely used in a MSA.

3.2 Authentication and access control in microservice architecture

For a MSA, authentication and access control are complex compared to a monolithic architecture because there are more services, and each service

needs its own authentication module. Furthermore, the MSA might have different types of APIs where the access policy differs. The same MSA could for example have a public-facing API, a private API and a separate API exposed only to specific business partners [9].

3.3 Service discovery

According to F. Montesi and J. Weber [12], as microservices are commonly deployed in a cloud-environment, the network addresses of the services are not necessarily statically known. Hence there is a need for the participants of the MSA to be able to discover the location of the services. This is the problem addressed by the service registry. The services uses the service registry to publish their location and when requesting a service, the client queries the service registry to get the location of the service. Furthermore, the service discovery can be implemented differently depending on the design pattern of the MSA. More precisely, this means the service registry could either utilize server-side discovery or client-side discovery.

3.4 Design patterns

In order to construct a MSA, decisions regarding the design choices need to be taken. To ease this procedure and to create a robust design, several design patterns for different problems and needs have emerged.

3.4.1 API Gateway pattern

The API Gateway pattern is described in [12] as a design pattern used to address the problem of multiple calls invoked by different channels to multiple microservices. The different channels might use different protocols and the consumers might need different formats of the responses. For example, a mobile client (e.g. a smartphone) and a web browser might have different needs and capabilities. To address this, the API Gateway is implemented as the single entry point to the MSA. It acts as a proxy to provide multiple APIs for several types of clients with potentially aggregated responses. The API Gateway can also be responsible for load balancing, service discovery, monitoring, and security. However, it should be noted that if the API Gateway becomes the single-point-of-decision it might violate the ”defense in depth” principle [8].

3.4.1.1 Authentication and Authorization

In the API Gateway pattern, the authentication and authorization of the client happen at the API Gateway. In this way, the API Gateway becomes an essential component for centralized authentication. Common solutions for implementing authentication and authorization towards the client at the API Gateway is by federated identity [13]. The authentication could for example be implemented by utilizing the OAuth2.0 protocol along with JSON web tokens (JWT) for authorization towards the microservices [13, 9].

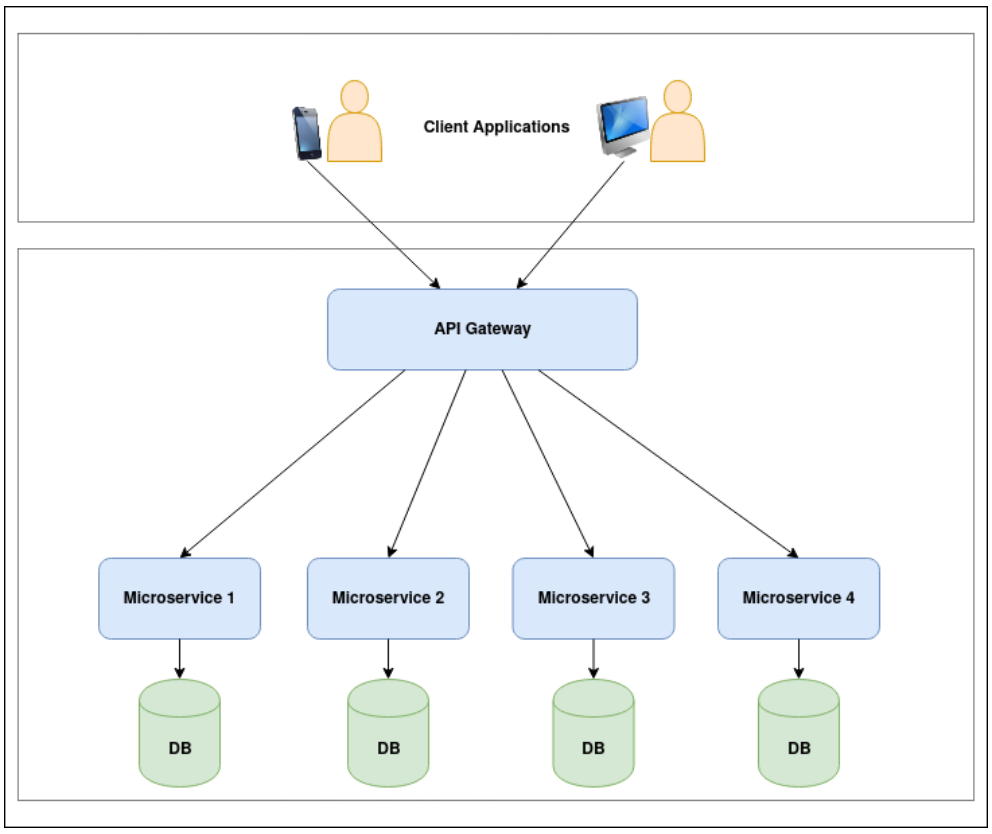

Figure 3.1: API Gateway pattern

Figure 3.1 shows an overview of the API Gateway pattern. The clients interact with the API Gateway which in turn proxies the requests to the underlying microservices in the system.

3.4.2 Sidecar pattern

The Sidecar pattern [14], sometimes referred to as sidecar proxy, is a design pattern that addresses the services needs for related functionality such as networking, monitoring, and logging. This is accomplished by attaching a separate process, the sidecar, to the parent service. In this way, the related functionality is isolated and encapsulated from the service's core functionality. A sidecar is often used along with a service mesh [15]. The service mesh is an infrastructure layer to enable communication between services in an easy way by providing routing, internal load balancing, encryption, authentication, and authorization [9]. The service mesh can be divided into two different components, the control plane, and the data plane.

3.4.2.1 Control plane

According to G. Miranda [16] the control plane is what the user interacts with and can, depending on the implementation, be responsible for example routing to the internal services, user authorization, and rate limiting to mitigate denial of service attacks. The functionality provided by the control plane is not in itself needed for the service mesh to function, hence it is not mandatory to utilize a control plane in the service mesh. However, it is recommended for practical reasons as the control plane adds management abstraction otherwise needed by the data plane.

3.4.2.2 Data plane

The data plane is explained in [17] as the set of proxies deployed as the sidecars to the microservices. The responsibilities of the data plane are routing, authentication, authorization, health checking, load balancing, and providing observability of the traffic. These responsibilities are possible as data planes or specifically the proxies handle the movement of the data and thus have control of all the service-to-service communication over the network within the MSA. A simplified description of the data plane is that it is responsible for the management of the service-to-service communications, also referred to as east-west traffic, which is a crucial part of the behavior of the MSA as it relies heavily on communication. Typically the service mesh is provided as a software product with the goal to not requiring any changes to the application code. Hence the proxies in the data plane should be close to transparent to the attached microservice core functionality [16].

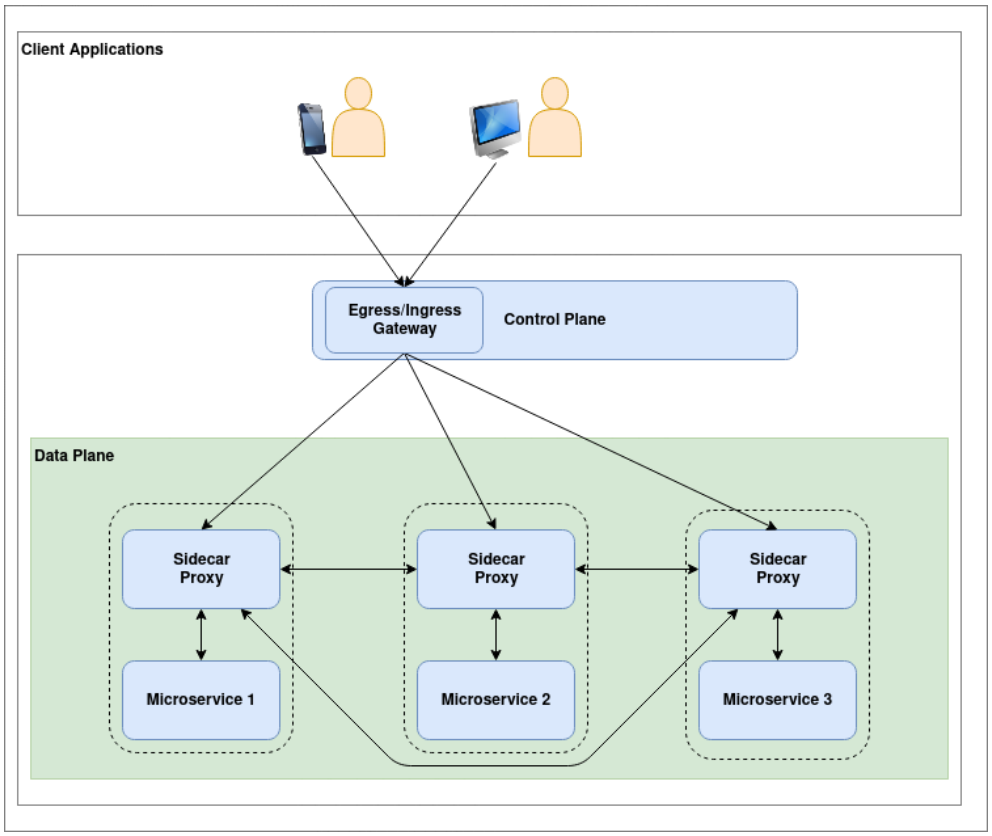

Figure 3.2: Sidecar pattern

Figure 3.2 shows an overview of the sidecar pattern. The microservices are isolated and only communicate directly with the attached sidecar which in turn further forwards the data to the other services.

Chapter 4

Threat modeling

According to A. Shostack [18], threat modeling is a practical method to find security-related problems in systems. The aim of threat modeling is to foresee threats to a system before it is built and gather knowledge regarding how the threats can affect the system. This is done by addressing four questions:

1. What are you building?
2. What can go wrong?
3. What should you do about those things that can go wrong?
4. Did you do a decent job of analysis?

The first question addresses the architectural design to increase knowledge of what type of data and how data flows through the system. It can also answer questions regarding where the trust boundaries in the system are and who is in control of these, for example, if some servers are hosted at a data center offsite. This can be communicated in a diagram which provides an intuitive way to see where the major potential threats to the system are, e.g. threats crossing the trust boundaries are probably important.

The aim of answering the second question is to identify threats to the system. To identify threats frameworks such as STRIDE or attack trees.

4.1 STRIDE

In 1999, Loren Kohnfelder and Praerit Garg invented the threat modeling approach STRIDE. It is an acronym for Spoofing, Tampering, Repudiation, Denial of Service, Information Disclosure, and Elevation of Privilege.

STRIDE is a framework originally invented with the purpose of identifying types of attacks that might occur towards software. Each word in the acronym violates a necessary property of a software [18]. That is:

- **Spoofing** violates authentication;
- **Tampering** violates the integrity;
- **Repudiation** violates non-repudiation;
- **Information disclosure** violates confidentiality;
- **Denial of Service** violates availability;
- **Elevation of privilege** violates authorization;

In some cases, phrases such as ”STRIDE categories” or the ”STRIDE taxonomy” are used. However, A. Shostack emphasizes that the aim of STRIDE is to be a helpful tool for finding attacks and not to categorize them, as a single attack can violate multiple security properties.

4.2 Attack Trees

In 1999 B. Schneider explained the method of attack trees [19]. According to B. Schneider, attack trees are a representation of the security in a system along with its subsystem. The trees are structured to make it possible to store them in a database to be reused with the collected expertise of the system. Hence attack trees can be used both in order to make security decisions and also to understand how new attacks will affect the security of the system.

The representation of the tree can contain both the attack and the countermeasures where the root of the tree is the goal of the attacker and the leaf nodes represent the actual attacks. Furthermore (parent) nodes can either be labeled as ”or” nodes or ”and” nodes. For the case of ”or” nodes, it means that any of the children provides a way to the goal i.e. the parent node. For ”and” nodes, on the other hand, all the children represent sub steps in the process of reaching the goal, thus all the sub steps need to be completed to reach the parent node. For leaf nodes, a set of values can be assigned which is associated with the attack. Examples of values could be boolean values, such as if an attack is possible or not or if special equipment is needed to carry out the attack. The values can also be continuous such as time, monetary cost, or cost in terms of resources in order to carry out the attack. Lastly, the nodes can be assigned with countermeasures to mitigate the attack.

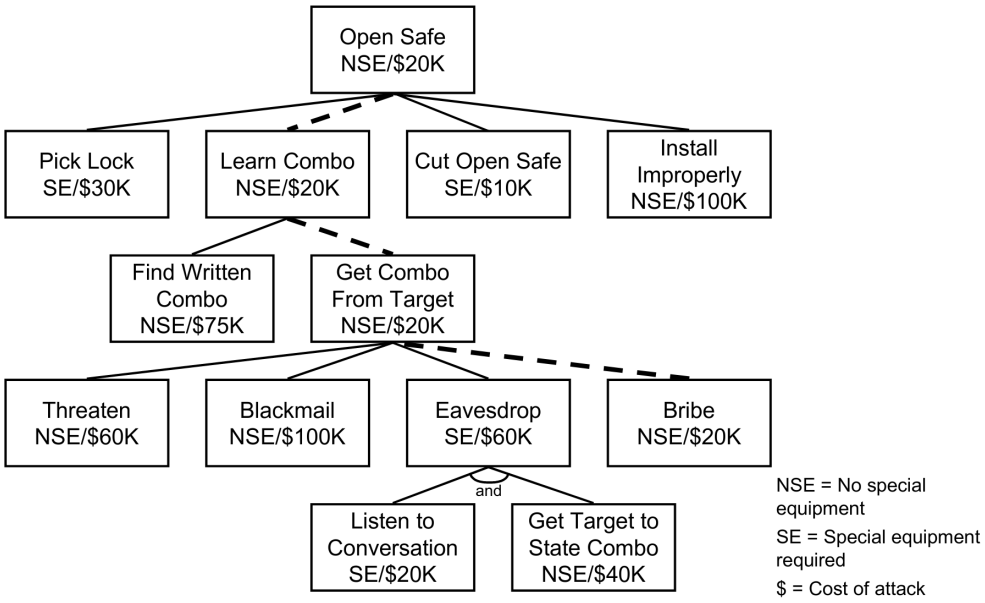

Figure 4.1: Attack tree showing the cheapest way to compromise a safe without special equipment. [19]

Figure 4.1 shows a visual representation of an attack tree where two values have been assigned to the nodes. One of which is the boolean value for whether special equipment is needed or not and the other value represents the monetary cost to perform the step. The dotted lines represent the cheapest path for the attacker without the use of special equipment.

A. Shostack [18] describes attack trees as an alternative method to STRIDE which can be utilized for identifying threats in a system. A. Shostack argues that even for security experts it is challenging to create new trees. However, since attack trees can be reused it is possible that there exists a relevant attack tree for the system or parts of the system which can be used in order to identify threats. As the attack trees are hard to create there are a set of issues related to the subject such as completeness, scoping, and meaning.

Completeness is an issue because if essential root nodes are missed during the construction of the attack tree, a set of potential attacks will be left out as a result. Furthermore, it is hard to know how many nodes a branch of the tree should have before it is complete. Hence it cannot be known whether the attack tree is complete or not. Scoping might be an issue because some threats are coupled with underlying technologies, and thus will be categorized as impossible to fix. Meaning is an issue because it is time-consuming to understand new trees as there are not necessarily any consistency regarding

”and” vs ”or” nodes in the tree.

4.3 Attack graphs and attack simulations

Attack graphs were explained by C. Philips and LP Swiller [20] as a method to model network risks as a graph where the states of the possible attacks are represented as the nodes in the graph and the edges represent the transition from one state to another. They emphasize that the edges can represent both transitions made directly by the attacker or by an action as a consequence of deceiving a user to make the transition. It should be noted that this is different from attack trees where the leaf nodes represent the attack while in attack graphs the attack is represented by the edges, transitioning from one state to another by modifying the state. Regarding the states (i.e. nodes) they explain that a state can contain for example the access level of the user and the current implications of the attacks.

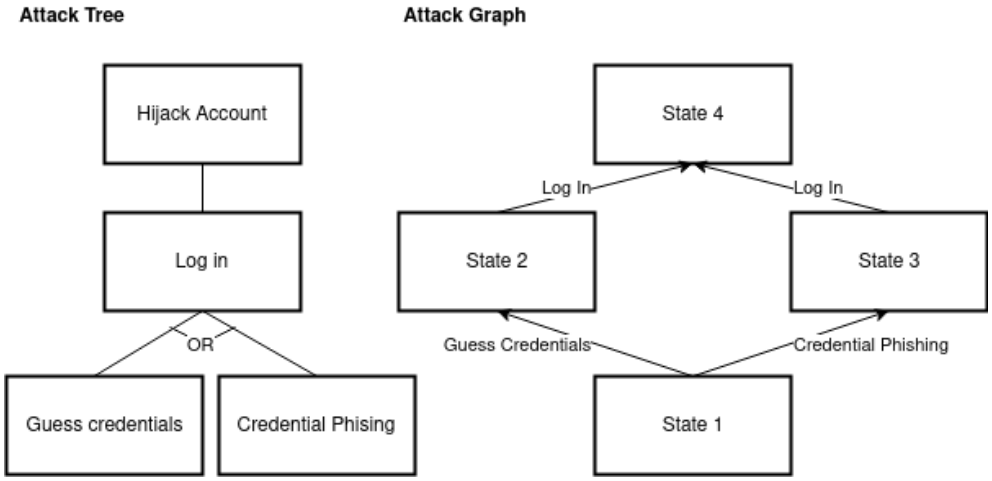

Figure 4.2: Attack tree (left) and attack graph (right) where the goal of the attacker is to hijack an account.

In order to highlight the differences between attack graphs and attack trees, Figure 4.2 provides a simplified example of a scenario for an attacker to reach the goal of hijacking an account. The figure to the left displays an attack tree and the figure on the right displays an attack graph for the same scenario.

The edges in the attack graph can be assigned weights, for instance, probabilities or probability distributions which can be used along with graph algorithms. This can further be used in order to perform attack simulations, by

simulating an adversary attacking the system to gain insights regarding where the system is the most vulnerable [21].

4.4 Meta Attack Language (MAL)

The meta attack language (MAL) presented in [21] is a language developed with the aim to provide a formalism for security experts to design domain-specific languages. With the use of MAL attack graphs for a specific infrastructure can be generated, and the attack graphs can later be used in order to perform attack simulations on the infrastructure. The attack simulations can be described as a set of virtual penetration tests used to identify security weaknesses in the system. The attack simulations result in the global time to compromise the systems. That is, the time it will take for an adversary to compromise the system which is used under the assumption that a rational adversary will take the shortest path to compromise the system. MAL is explained in detail in [21], however, a summary of the paper is given in the following section.

4.4.1 Formalism of MAL

The formalism of MAL consists of objects (x), classes ($\mathbf{X}$), attack steps (A), links (λ), associations ($\mathbf{\Lambda}$) and roles ($\mathbf{\Psi}$). P. Johnson et al. [21] exemplifies this by the use of physical objects, the cullinanDiamond and the antwerpVault belongs to the classes Jewelry and Vault respectively. That is, $cullinanDiamond \in Jewelry$ and $antwerpVault \in Vault$. Classes have associated attack steps such as $Jewelry.steal$ or $Vault.open$ which can either be of OR type or AND type. Furthermore each attack step is associated with a probability distribution P for the expected time, ϕ, it takes to complete the step called the local time to compromise (T_{loc}). Hence the expected time to complete the attack step A is is denoted $\phi(A) = P(T_{loc} = t)$.

The classes are denoted as the set $\mathbf{X} = \{X_1, ...X_n\}$ where each class X_i with the set of associated attack steps A are denoted $A(X_i)$ or X.A. Hence if the attack step A on class X is of type OR it is denoted as $t(X.A) = OR$ and if the expected time to complete A on X is given by a Gamma distribution with mean 12 and standard deviation of 6 hours it is denoted $\phi(X.A) = Gamma(24, 0.5)$. Mandatory, there is the class Attacker, $\Xi \in \mathbf{X}$, with the only attack step $\Xi.\xi$ to mark the entry point of the attacker.

To describe relations between objects, e.g. if the antwerpVault contains a cullninanDiamond, it is represented as a binary tuple $\lambda = (x_i, x_j)$ where λ is

the link between the objects x_i and x_j. To describe the relationship between classes the set of associations, denoted $\mathbf{\Lambda} = \{\Lambda_1, ..., \Lambda_n\}$ are used. Hence a link λ can belong to an association Λ thus a relationship between classes can be denoted $x_i, x_k \in X_m, x_j, x_l \in X_n | \lambda_1 = (x_i, x_j) \in \Lambda \wedge \lambda_2 = (x_k, x_l) \in \Lambda$.

The aforementioned example that the antwerpVault contains a cullinanDiamond could generic for classes be exemplified as the Vault has a container role with respect to the Jewelry. The role $\mathbf{\Psi}$, a class $\boldsymbol{X}$ plays in an association $\mathbf{\Lambda}$ is denoted $\mathbf{\Psi}(X_i, \Lambda)$ or X.$\mathbf{\Psi}$ such as $Jewelry.container = Vault$.

4.5 CoreLang

In [22] S. Katsikeas et al. present the domain-specific language (DSL) coreLang. The language is based on MAL with the aim to provide functionality for modeling of domain-specific attributes in IT infrastructures and an analyzing tool for weaknesses associated to known attacks. To enable the modeling six classes (also referred to as asset categories) provided by coreLang are system, vulnerability, user, identity and access management (IAM), data resource, and networking. In the present version of coreLang the system category has been renamed to Compute resources [23].

4.5.1 Compute resources

The Compute resources category is the representation of compute instance and consists of the assets Hardware, Software product, Application, identity and prevention system (IDPS), and Physical Zone.

Hardware asset, as the name suggest, represent the hardware of the compute resource. It contains attack steps such as Supply chain attack, Hardware modification, Denial of Service (DoS).

Software product asset represents a particular application along with its related vulnerabilities. The asset is an extensions of the Information asset in the data resource category and exists for convenience. Therefore the attack steps are the associations related to the application such as read, write, modify and deny. For example if the application is denied, other software dependent on the application should be affected.

Application is the asset to represent executable entities or entities that can execute applications, e.g. an arbitrary application or the operating system. Therefore the asset is complex and contains numerous attack

steps, some of the essential attack steps are local connect, network connect and authenticate.

Identity and prevention system (IDPS) is an extension to the application asset containing countermeasures and attack steps as result of by-passed countermeasures.

Physical Zone is an asset to represent the physical location of the compute resource. It contains a single attack step, gain physical access, which is an association to the hardware asset.

4.5.2 Vulnerability

The vulnerability category is the representation of software- and hardware vulnerabilities. The category contains abstract asset, vulnerability, along with two assets for software vulnerability and hardware vulnerability respectively.

Vulnerability is an abstract asset providing common attributes shared among software- and hardware vulnerabilities. The asset contains one countermeasure, remove, which represents the removal of the vulnerability. It also contains attack steps such as attempt abuse, abuse, attempt exploit, exploit and impact.

Software vulnerability as the name suggests is an asset to represent vulnerabilities in software. The asset contains a number of countermeasures to model whats is required to exploit the vulnerability for example network-, local- or physical access and if low or high privileges are needed. Two attack steps related to the exploitation of the vulnerability, exploit trivially and exploit with effort exists. The latter represent an exploit where effort is needed by the attacker, hence a exponential distribution is attached to the attack step. The asset also contains attack steps for read, modify and deny.

Hardware vulnerability represents the vulnerabilities affecting the hardware. The attack steps share similar characteristics with the attack steps in the software vulnerability asset.

4.5.3 User

The user category represent a user of the system which in it self is a attack surface for attacks such as social engineering or phising. The category contains of the single asset User.

User is the asset to represent a User of the system. The asset contains two countermeasures, no password reuse and security awareness. The attack steps are related to social engineering, phising, exploiting, reusing user credential and delivering removable media, e.g. a USB with malicious code, to the user.

4.5.4 Identity and access management (IAM)

The IAM category is a representation of the users roles in the system, i.e. identity, group and privileges. The category consists of the assets IAM Object, Identity, Privileges, Group and Credentials.

IAM Object is an abstract asset for common attack steps in the IAM category. The asset contains one countermeasure, disabled, which is used to model the probability of the existence of the IAM role. Furthermore the asset contains of three attack steps related to the attackers assumption of the IAM role.

Identity is an extension to the IAM Object which represents the IAM identity mapped to its privileges. The asset consists of three attack steps related to assuming the identities privileges and locking out the identity.

Privileges also extends the IAM Object and contains a single attack step for assuming the privileges of the role.

Group represents a group of identities or groups. Similar to Privilege, Group is an extension to the IAM Object and consists of a single attack step related to assuming the privileges.

Credentials is an extension of the Information asset in the data resources category. The assets represents the credentials associated to the access control for the Identity. The asset contains four countermeasures to model how hard it is for an attacker to obtain the credentials. Likewise, the asset contains numerous attack steps related to obtaining, reusing or writing the credentials.

4.5.5 Data resources

The category data resources contains two assets, information and data.

Information is the asset representation of the information which can be extracted from the data. The asset contains of numerous attack steps

regarding accessing, extracting, reading, writing, deleting or denying the information.

Data is an asset to represent data either at rest or in transfer. The attack steps concerns accessing, reading, writing, denying and deleting the data and gives the possibility to model whether the data is encrypted and signed or not.

4.5.6 Network

The network category represents all assets related to the network of the systems. The category consists of three assets, Network, Routing firewall and Connection rule.

Network is an asset to represent the OSI Model, hence the attack steps concerns attacks related to different layers in the OSI model, e.g. both attacks that require physical access and attack that intercepts the traffic. Example of attack steps are Denial of Service (DoS), eavesdropping, Man in the middle (MITM) and network forwarding. The asset also contains countermeasures such as network access control, eavesdropping- and MITM defense.

Routing firewall is an extension of Application in the Compute resource category and represents the firewall placed at the entry point to the connected networks. The asset contains two attack steps, Denial of Service (DoS) and full access. The DoS step cascades the DoS to connected hosts and full access indicates that the firewall has been compromised.

Connection rule is the asset representing the rules of the firewall both for the network and the application. The asset have two countermeasures, restricted and payload inspection which are related to detecting and preventing network attacks. The attack steps of the asset are related to accessing the Network or Application and Denial of Service (DoS).

4.6 SecuriCAD

In order to generate visual representations of attack graphs, one can utilize existing tools. One such tool is SecuriCAD, a product developed by the company foreseeti [24]. SecuriCAD is based on DSLs and compatible with

languages such as MAL and coreLang to perform attack simulations of the attack graph along with visual representations of the attacker model. Hence a model along with its assets and associations can be implemented in CoreLang (or MAL) and imported to SecuriCAD to perform the attack simulations and produce a graphical representation of the attack steps.

Chapter 5

Threat modeling in microservice architectures

Microservices are distributed and rely heavily on communication, hence a set of possible attacks toward the MSA are also related to the underlying technologies such as hardware, cloud environment, containers, and network. To understand the possible attacks to a MSA, one can divide the attacks based on the different layers. T. Yarygina and A. H. Bagge [25] divide the MSA into six layers, hardware, virtualization, cloud, communication, service/application and orchestration. Among the six aforementioned layers, the communication, and the service/application layers and virtualization were chosen as the most relevant for the scope of this project. Section 5.1 presents the layer of communication, Section 5.2 presents the layer of the individual services and Section 5.3 presents the layer related to virtualization.

5.1 Communication attacks

Microservices communicate with each other via messages over a network. Therefore one attack vector is the communication channels for sending messages. Performing network attacks toward the communication is similar to traditional network attacks [25]. In the following subsections, network attacks relevant to this project are summarized.

Denial of Service (DoS) attacks are attacks on the availability of resources in the network. Resources could for example be web- or email servers. DoS happens when an attacker sends large amounts of data targeted to specific resources the network relies on. When the transmitting data

exceeds the capacity of the bandwidth in the network, consequently exhausting the network. If the traffic in the attack is generated by multiple parties, the attack is called a Distributed Denial of Service (DDOS) [26].

Man-in-the-middle (MITM) attacks occurs when an adversary manage to position itself between hosts on the network intercepting the traffic. The adversary can then eavesdrop the traffic or perform transmitted data manipulation by altering the data [27].

ARP Spoofing occurs when an attacker forges ARP packets on the network with the goal to masquerade as another host. This is possible as the hosts on the network will cache the ARP replies since the ARP protocol is stateless [28]. As a result of a successful ARP spoofing attack, the packets meant to the victim will be sent to the attacker instead.

Identity spoofing occurs when an attacker masquerade as another host by using the identity of the host for example by using stolen authentication credentials. As a consequence the attacker may be able to send messages which appears to come from the victim [29].

Eavesdropping might occur if the network traffic is not encrypted or if the encryption is flawed. An attacker connected to the network can eavesdrop the communication [30].

Replay attack is an attack on the underlying security protocol used in the network. It occurs when an adversary replay messages deceiving the victim into thinking the protocol is completed [31]. For example, a replay attack could be if an attacker eavesdrop the hash of a password sent to a server and then sends the hash tricking the server that the attacker knows the password.

Transport Layer Security (TLS) attacks: TLS is a protocol to provide cryptography on the application layer in the network. TLS is for example used by HTTPS. There are multiple attacks on TLS, some attacks consists of exploiting known vulnerabilities such as Heartbleed [32] and other attacks such as POODLE [33] exploits the protocol itself. For brevity no in depth description of all known attacks will be explained here, but IETF has published a summary of known attacks on TLS [34].

5.2 Service attacks

Possible attacks toward the microservices are similar as to web servers with common attacks such as cross site scripting (XSS), Cross-Site Request Forgery (CSRF), encoding and serialization attacks, HTTP verb tampering and injections [9]. OWASP publishes a list, OWASP top 10, of common attacks against web applications summarized below.

Broken Access control: Attacks on the access control occurs when an attacker are able to act outside of its intended permissions. This could for example be done by exploiting vulnerabilities related to tampering with URL parameters, HTTP verbs, JWT or cookies [35].

Cryptographic Failures: Exploiting vulnerabilities related to cryptographic failures can happen when the cryptographic methods used are flawed, weak or non existent. This concerns both data at rest and data in transit. Examples of cryptographic failures that could result in exposure of sensitive data includes use deprecated hashing algorithms such as SHA1 or MD5, randomness used for cryptographic algorithms are not cryptograpically secure or default cryptograpic keys used or stored along with the source code, error messages or other side channel information exposed [36].

Injection: An application is vulnerable to injection attacks if it executes untrusted commands such as executable code or OS commands provided by the user without server-side validation. Common injections includes SQL-injection, NoSQL-injection, Cross-site scription (XSS), among others [37].

Insecure design: Attacks as a result of insecure design are attacks that target the architectural flaws rather than the implementation of the technologies. Examples could be bots purchasing and reselling popular items from an e-commerce site due to the lack of anti-bot design. As a result this can harm the reputation of the company [38].

Security Misconfiguration: Attacks as a result of insecure design are attacks that target the architectural flaws rather than the implementation of the technologies. Examples could be bots purchasing and reselling popular items from an e-commerce site due to the lack of anti-bot design. As a result this can harm the reputation of the company [38].

Vulnerable and Outdated Components: Exploiting vulnerable and outdated components means targeting components of the software with known vulnerabilities. This could for example be third-party libraries that are no longer maintained with known vulnerabilities or older, unpatched, versions of components. Exploiting such vulnerabilities could severely affect the security as they typically run with the same permissions as the application [39].

Identification and Authentication Failures: Attacks towards identification and authentication as the name suggests exploits weak authentications mechanisms in the software. Example of such attacks could be credential stuffing attacks, brute force attacks, dictionary attacks or exploiting long set session timeouts [40].

Software and Data Integrity Failures: Attacks related to software and data integrity failures exploits applications that relies on untrusted third-party libraries, repositories or plugins without properly verifying their integrity. A possible attack scenario could be for an adversary to push malicious code to the library or repository the application relies on. Related to this are also exploiting an applications use of insecure deserializations where an adversary can modify the encoded or serialized object [41].

Security Logging and Monitoring Failures: Regarding security logging and monitoring category in OWASP top 10, there are no attacks as a direct consequence of logging and monitoring but instead the lack thereof. Logging and monitoring is crucial for detecting and responding to data breaches. The lack of logging and monitoring enables an adversary to run automated attacks without being detected [42].

Server-Side Request Forgery (SSRF): Server-side request forgery occurs when an adversary exploits a web application fetching remote resources. An example of such attack could be that the adversary sends a crafted URL, later to be requested but not validated of the server. The crafted URL could for example point to local resources, accessing data from the cloud environment or be used to further attack internal services [43].

5.3 Virtualization attacks

In [25] the category for virtualization attacks covers both hypervisor- and container-based virtualization, however for the scope of this thesis only container-based attacks will be covered. Attacking the container-based virtualization tries to exploit the containers along with their isolation by exploiting vulnerabilities in container runtime, image vulnerabilities, image misconfigurations, or the use of untrusted images [44]. The following subsections contain a summary of possible attacks towards the container-based virtualization.

Remote code execution (RCE) in the context of container-based virtualization exploits image vulnerabilities. A remote code execution occurs when an attacker is able to inject code later to be executed by the application [45]. For the case of Docker containers, if an attacker has the ability of writing to the Dockerfile a remote code execution can occur by writing arbitrary commands to the ”RUN” command which consequently will be executed by Docker [46].

Unauthorized access Unauthorized access is an effect of image misconfiguration and can occur if an application running in the image does not utilize the principle of least privilege. If the application runs with root privileges it has possibilities to take full control of the container [44].

Network-based intrusion can also be a result of image misconfiguration and can occur when unnecessary ports such as SSH, are left open in the container. An adversary could take advantage of this in order to gain remote access to the container [44].

Compromising secrets: If secrets such as database connection parameters are stored in the container it can lead to undisclosed read and/or tampering with the database [44]. Furthermore the application might use credentials in order to get access remote APIs, if such secrets are compromised it might give the attacker permission to interact with the restricted resources [46].

Container malware: Similar to a regular operating system a container might be targeted with malicious code such as viruses, worms, ransomeware or trojans. Furthermore if a container uses untrusted images additional threats for example that the image might have preinstalled backdoors also exists [44]. On the application layer it is common that third party

libraries which can easily be installed with language-specific package-manager such as Maven, pip or npm. However like any other software, the third party software might contain vulnerabilities [46].

Privilege escalation and container escapes: Container escapes occurs when vulnerabilities in the container runtime are exploited and the adversary manage to break out of the sandbox environment. One such vulnerability is CVE-2017-5123 in which applications can modify the capabilities of Docker runtime [44]. Another is the RunC container escape listed as CVE-2019-5736 [47], which allows an adversary to break out of the docker container and gain root access on the host system.

Chapter 6

Method

In this chapter, an overview of the method used for this project is given. The chapter begins with a description of the research process in Section 6.1. Section 6.2 presents an overview of the evaluated MSA and Section 6.3 describes the attack scenarios the design patterns were evaluated on. Section 6.4 describes the measurements used to evaluate the risks within the design patterns and finally, in Section 6.5 the reliability and validity of the method is given.

6.1 Research Process

In this project, threat modeling and attack simulations were used in order to evaluate security threats and compare the risks between design patterns used in MSAs. For the implementation phase, the process can be divided into three parts:

1. Identify threats
2. Construct models
3. Perform attack simulations
4. Analyse the results of the attack simulations

The first part was first and foremost addressed in the literature review for threats related to MSAs in general. To identify threats to the MSAs under evaluation STRIDE was used as a complement to the literature in order to choose relevant assets and attack scenarios by taking the STRIDE properties into consideration for the simulated application.

The second part was to construct the asset model for each design pattern. The asset models were implemented using SecuriCAD running CoreLang as the underlying domain specific language. No additional assets were required for the project, thus no changes were made to CoreLang. In order to compare the risks between the design patterns, the local time to compromise was set at this stage. The local time to compromise is further addressed in Section 6.4.

The following part after constructing the asset models was to perform the attack simulations for the chosen attack scenarios. The simulations were performed in SecuriCAD generating the attack graphs and risk values for the simulated attacks which were used during the analysis.

6.2 Evaluated microservice architecture

In order to compare the design patterns, a typical design for a MSA was needed in a minimal manner to showcase the functionality provided by the API Gateway pattern and the sidecar pattern respectively. Except for the architectural differences between the design patterns the overall MSA had to be identical in order to limit sources of errors or irrelevant factors to affect the result. For the case of this project, the MSA to evaluate was decided to be a chat application. The chat application consists of two microservices, one authentication service, and one chat service.

The purpose of the authentication service is to handle functionality related to the authentication for the users of the system along with storing user information and credentials. The authentication service consists of an API along with a database. The API simulates a restful API to handle functionality for creating new users, validating users, and providing authorization for the stored user information. The API is connected to its own database where data at rest regarding the user information is stored.

The purpose of the chat service is to handle functionality related to the exchange of messages between two or more parties such as sending and retrieving messages. Similar to the authentication service, the chat service consists of an API along with a database where the API simulates restful communication for the chat. The API handles functionality for posting new messages and fetching messages between two users. For the data at rest, the messages are stored in the chat database which contains the messages along with metadata and information regarding to whom the message was addressed to, and who sent the message.

In order to be able to provide the chat functionality, the chat service has to communicate with the authentication service, or in other words, east-

west traffic is needed in the architecture. This is because the authentication service should be able to provide an authorization mechanism regarding the participants' relationship, for example, if user A is allowed to send messages to user B.

6.3 Attack scenarios

In order to be able to evaluate the design patterns as a whole, a set of attack scenarios were evaluated on the respective model. For the scope of this project, one attack scenario for each of the categories of communication attacks, service attacks, and virtualization attacks was performed.

6.3.1 East-west traffic eavesdrop

The communication in the modeled MSA consists of both north-south (client-to-server) communication and east-west (service-to-service) communication. This is common to any MSA even though it depends on the requirements of the application. However, while dividing the application into smaller services the application naturally depends more on east-west traffic compared to its previous monolith. Furthermore, the communication between the services is more likely not to cross any trust boundaries in contrast to north-south traffic thus it is reasonable to assume that less effort will be taken in order to protect this traffic. Hence the communication attack on the MSA was chosen to be eavesdropping on the east-west traffic.

Pre-condition: The state of the attacker is similar to any benign user before the attack is carried out as the attacker is able to interact with the entry point of the MSA but without any special privileges. After recognisance the attacker might know the system is a MSA but is lacking deeper knowledge regarding the system.

Post-condition: After successfully performing the attack the adversary would be able to intercept the traffic between the chat service and the authentication service and steal the data in transit. Depending on the implementation, the intercepted data could either be encrypted or not.

6.3.2 Compromising secrets

For the attack scenario on the container-based virtualization, compromising secrets were chosen as the goal for the attacker. Compromising secrets is not

an attack targeted towards microservices specifically but any system with the requirement of storing tokens for authentication towards remote systems such as a database or API-keys for a remote service. This is a common requirement needed by microservices as they rely heavily on communication thus API-keys are needed to communicate with internal and external services. Furthermore, if each service needs a set of secrets, naturally as more services are added the number of secrets increases and can thus be harder to control.

Pre-condition: Similar to the eavesdrop attack, the state of the attacker before the attack is the same as for a benign user without any special privileges.

Post-condition: After successfully compromising the secrets the attack would be able to use them to connect to the database with the same privileges as its corresponding service.

6.3.3 Broken access control

For the attack scenario of the individual microservices, broken access control was chosen as it is the most occurring security flaw for applications in OWASP Top 10 from 2021. Bypassing access control can have a wide range of consequences and can be targeted to different parts of the system. For the scope of this thesis, in order to evaluate broken access control, the goal of the attacker was to be able to read from the database containing the user information.

Pre-condition: The state of the attacker before the attack is carried out is the same for the attacker as a benign user without any special privileges. The attacker can sign up for an account on the service and should thus be able to read the attackers own user information provided by the application.

Post-condition: After successfully being able to bypass the access control the attacker is able to read the data in database. The data might either be encrypted or not but the attacker is able to steal it and process it on their own machine.

6.4 Risk measurement

In order to measure the security risks for the evaluated models the global time to compromise (GTTC) was used. The GTTC is a measure of how long it

takes for the attacker to reach the goal of the attack given by the expected time it takes to compromise each attack step, i.e. the local time to compromise (LTTC), in the attack graph. In other words, the GTTC is given by the sum of all LTTC chosen by the attacker in the attack graph

$$GTTC = \sum_{i=1}^{N} LTTC_i \tag{6.1}$$

where N is the number of attack steps for the chosen path of the attacker in the attack graph. As $LTTC_i$ is given by the expected time it takes to accomplish the attack step i, each attack step i is associated with a probability distribution. In SecuriCAD, the probability distribution can be set by the user of the program. In order to minimize assumptions regarding the system, the probability distribution was set to an equal probability distribution for each attack step in the model.

The actions of a perfectly rational attacker would be to take the least time-consuming path in order to reach the goal. More specific it is reasonable to assume that the attacker will try to minimize the GTTC. Furthermore, the lower the GTTC is, the higher the likelihood is for the attacker to try to compromise the system on the given path. Hence a lower value for the GTTC of an asset indicates a greater risk of that asset to be compromised. In this way, the risks between the asset models can be compared. For example, if the goal of the attacker is x for asset models A and B where $GTTC_A(x)$ and $GTTC_B(x)$ denote the GTTC for the attacker to reach goal x in the respective model, the result of

$$GTTC_A(x) < GTTC_B(x) \tag{6.2}$$

would indicate that model A has greater risk than model B for the goal of x.

6.4.1 Software

In order to implement the asset model and perform the attack simulations SecuriCAD was used along with CoreLang as the domain specific language. The implementation was created on a machine running Fedora Linux as operating system. The following table shows the software along with their respective versions used in the project.

Software	Version
Fedora Linux	35
CoreLang	0.5.1
SecuriCAD Professional	1.6.4

6.5 Method validity and reliability

The project uses threat modeling along with attack simulations in order to evaluate the risks of the design patterns. Section 6.5.1 evaluates the action taken to ensure the validity of the method and Section 6.5.2 presents a discussion regarding how the reliability of the method can be assured.

6.5.1 Validity of method

To address the validity of the method it is of essence to highlight that the intention of the project is to evaluate the risks associated with common design patterns utilized in microservice architectures and not any specific technology or implementation of the patterns. In order to ensure the validity of the method, the asset models were implemented with careful consideration to replicate each design pattern as closely as possible. Additionally, the attack paths generated from the attack simulations were analyzed independent of the quantitative measures used to compare the risks between the patterns to validate that the attack steps correspond to a realistic scenario. Finally, external sources were used to identify the security challenges faced by the respective design patterns to investigate if the attack simulations correlate to the challenges.

6.5.2 Reliability of method

The field of cyber security is rapidly evolving where new exploits, as well as mitigations, are frequently developed. It is therefore possible that new threats will be discovered changing the threat landscape and thereby also the risks of the evaluated design patterns. Furthermore, the experiment uses attack simulations based on probability distributions thus there is a possibility that different attack simulations yield different results. Finally, as the project uses threat modeling along with attack graphs, there is a possibility that the final attack graph does not cover all possible attack steps and thus possible attack paths might unintentionally be excluded from the result.

Chapter 7

Implementation

The threat modeling was performed by constructing two different asset models, one for each of the design patterns which would be used to perform the attack simulations on. The asset models were implemented to simulate the application described in 6.2 utilizing the design patterns of the API Gateway pattern and the sidecar pattern respectively. The chapter starts with Section 7.1 where a description of the implemented asset model utilizing the API Gateway pattern is given and in Section 7.2 a description for the implemented asset model utilizing the sidecar pattern is given.

7.1 API Gateway pattern asset model

Figure 7.1 shows the CoreLang asset model representation of the API Gateway pattern implemented in SecuriCAD. The asset model is implemented following the characteristics of Figure 3.1 where the API Gateway is placed at the edge of the system to proxy the requests to the underlying microservices. As a result, the users of the system does not interact with the microservices directly, instead, all the request will pass through the API Gateway. Moreover, each microservice in the asset model contains its own database in accordance with the aforementioned figure. On the other hand, in contrast to the figure, the asset model only contains two microservices and provides a more detailed representation of the components and the communication within the system.

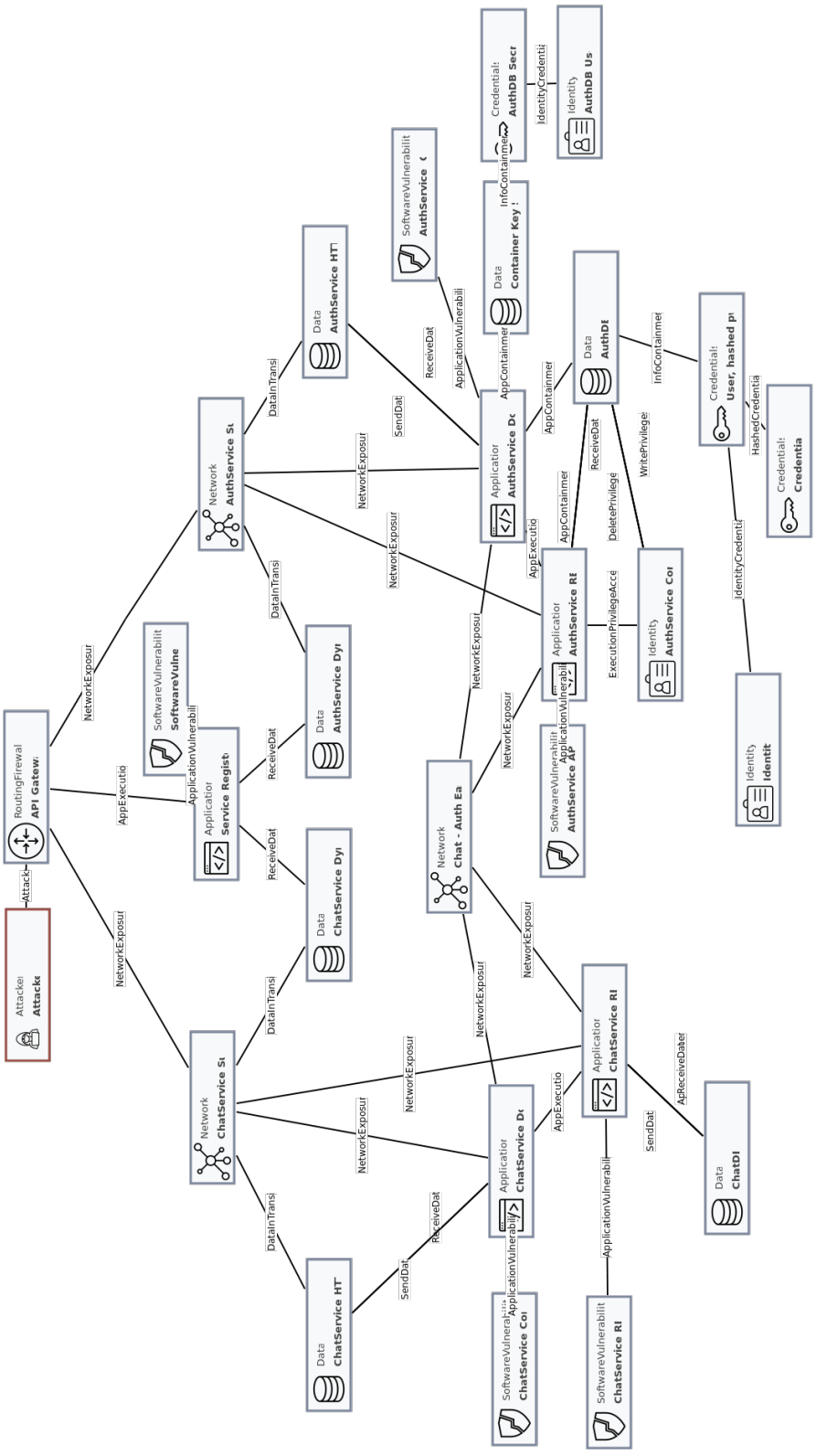

Figure 7.1: Asset model for API Gateway pattern

The entry point to the applications is the API Gateway which is represented as a *RoutingFirewall* in CoreLang, this is also the entry point for the attacker. The API Gateway is associated with each of the separate sub-networks for the ChatService and the AuthService respectively represented by the *Network* asset. The API Gateway is also associated with the Service Registry utilized by the microservices for pushing IP addresses to later be retrieved by the API Gateway.

The ChatService sub-network is associated with the ChatService Docker Container and the ChatService REST API, both of which are represented as an *Application* in CoreLang. In a realistic scenario though, the REST API is not directly exposed to the network, instead, there is a port forwarding through the Docker container. However, for the sake of modeling, this was depicted as a direct association to separate the scenario when a client is interacting with the Docker container from when the client is interacting with the REST API. The data in transit for the ChatService sub-network consists of HTTP requests and responses.

The ChatService Docker Container is associated with a *SoftwareVulnerability* to represent if the container is misconfigured or contains a zero-day vulnerability. Likewise, the ChatService REST API is also associated with a *SoftwareVulnerability* to represent vulnerabilities introduced by the developer such as bugs in the source code or the use of vulnerable third-party libraries. It is also possible to add countermeasure to assets within the model. However no countermeasures were enabled for any of the design patterns during this project as the main focus was on the robustness of the design patterns rather than specific technologies. For data at rest, the ChatService REST API is associated with the ChatDB which is a *Data* asset to for storing messages between the parties.

The east-west traffic between the ChatService and the AuthService concerns communication attacks and is thus represented as a *Network* asset. The asset is associated with the ChatService Docker container, the ChatService REST API, the AuthService Docker Container, and the AuthService REST API since the aforementioned assets will be exposed to the network in a similar way as the ChatService is exposed to its sub-network.

Regarding communication, the AuthService is similar to the ChatService exposed to its own sub-network with corresponding data in transit, but the microservice has more details as it was evaluated for the attack scenario specific to microservices. The AuthService Docker Container is an *Application* asset associated with the AuthService REST API by *AppExecution*. The Docker container is also associated with a *SoftwareVulnerability* and the AuthDB *Data*

asset to represent a database of the users mounted as a volume in the container. The AuthDB is associated with the AuthService REST API by *SendData*, *RecieveData* and *Appcontainment*. The SendData and RecieveData represent the open connection for querying the database and the AppContainment the association that the AuthDB contains the user information related to the AuthService REST API.

The AuthService Docker Container has also associated with another *Data* asset, the Container Key Store, this asset represents the storage of secrets and environment variables. Hence the asset is associated by *InfoContainment* with the *Credentials* asset AuthDB Secret. The AuthDB Secret represents a token needed in order to connect to the database and is thus associated by *IdentityCredentails* to the *Identity* asset which represents a user who can manage the AuthDB.

7.2 Sidecar pattern asset model

Figure 7.2 shows the CoreLang asset model representation of the sidecar pattern implemented in SecuriCAD. The asset model follows the characteristics of Figure 3.2 and contains a control plane at the edge of the system, routing the traffic to the sidecar proxies. In accordance with the aforementioned figure, the only direct communication connection each microservice has is with its attached sidecar which in turn communicates further with the different components in the system.

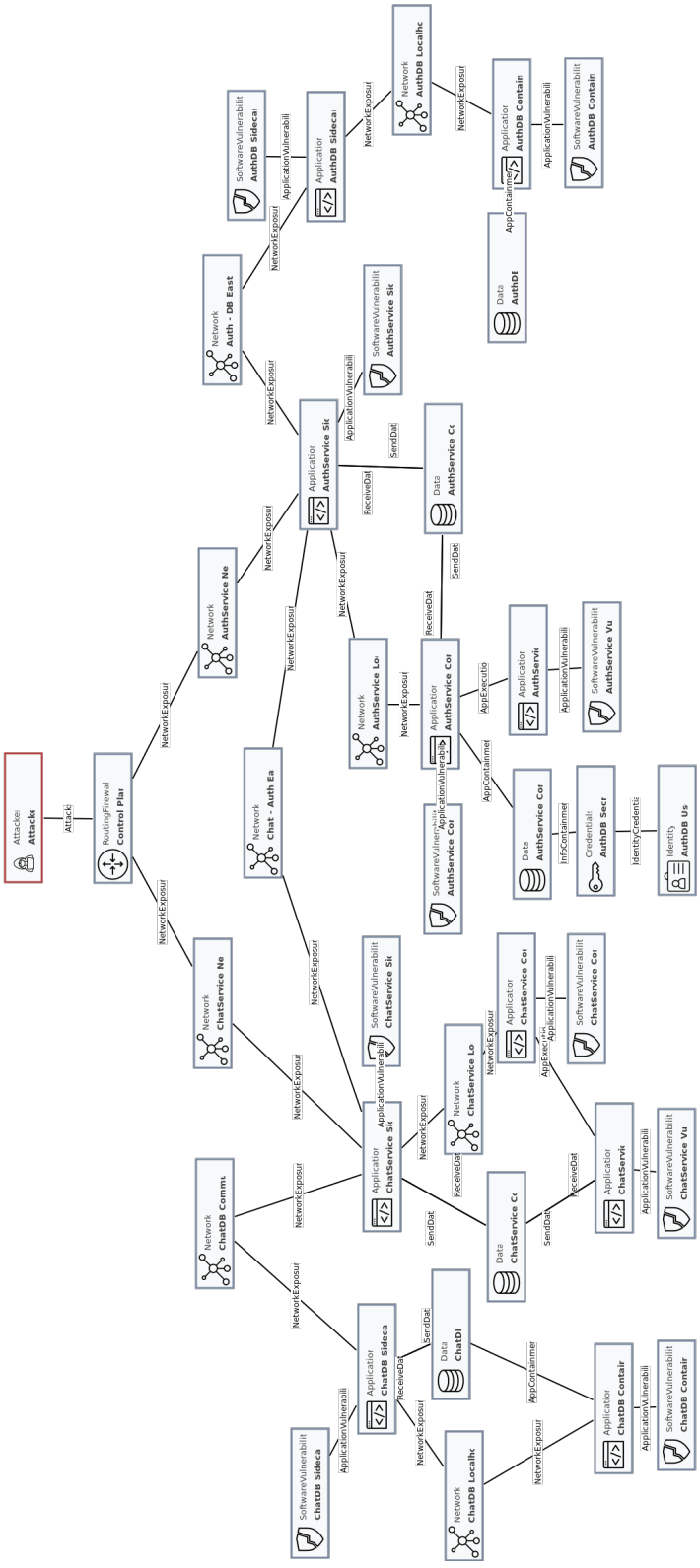

Figure 7.2: Asset model for Sidecar pattern

The entry point for the attacker is through the *RoutingFirewall* asset Control Plane. Similar to the API Gateway model the architecture is split up into different sub-networks for the different microservices. However, for the sidecar pattern, the ChatService only contains one direct association with the ChatService sub-network, namely the *Application* asset ChatService Sidecar proxy. This is because all the network communication from the ChatService will pass through and be inspected by the sidecar proxy. The ChatService Sidecar Proxy is associated with a software vulnerability to represent a zero-day vulnerability in the sidecar proxy. As the sidecar proxy and the ChatService runs in two separate containers they communicate with each other over localhost represented as a *Network* asset in the model.

The ChatService Docker container has two associations, the first is to a *SofwareVulnerability* asset and the second is an AppExecution association to the *Application* asset ChatService which represents the REST API. The REST API is further associated with a *SoftwareVulnerability* and the *Data* asset for the data in transit containing the HTTP communication data. Equivalent to the API Gateway pattern, no countermeasures were enabled for any asset in the model.

Regarding the east-west traffic between the ChatService and the AuthService it is represented as a *Network* asset for the same reason as in the asset model for the API Gateway pattern. However, as the communication between the services is handled by the sidecar proxies these are the only assets exposed to the network.

The AuthService sidecar proxy asset has similar to the corresponding asset for the ChatService six associations. The container running the sidecar proxy communicates with the AuthService docker container over localhost. The AuthService Docker container is similar to its equivalent in the model for the API Gateway pattern represented as a *Application* and associated by AppContainment to a key store represented as a *Data* asset. The key store has a InfoContainment association to the *Credentials* asset AuthDB Secret which is the authentication token needed for the AuthDB. However, in contrast to API Gateway pattern, the communication with the AuthDB is not directly associated with the AuthService Docker container or the REST API since the sidecar proxy is responsible for the communication. Instead, the querying of the database goes through the sidecar proxy for the AuthService to the AuthDB sidecar proxy.

The AuthDB sidecar proxy contains to additional associations. The first is to a *SoftwareVulnerability* and the second is a NetworkExposure to the *Network* asset representing the communication over localhost with the AuthDB Docker

container.

The AuthDB Docker container is the *Application* asset where the actual database for the AuthService is mounted. Hence it contains an AppContainment association to the *Data* asset AuthDB which is the data at rest containing information about the users of the application.

Chapter 8

Attack Simulation Analysis

In this chapter, an analysis of the attack simulations performed on the design patterns is given. The attack simulations cover a subset of the potential attacks toward MSAs. The result of the attack simulations is first presented and analyzed in Section 8.1 with respect to the global time to compromise. In Section 8.2 the attack graphs presented and analyzed for each attack scenario.

8.1 Risk Analysis

As described in Section 6.4 the time to compromise (TTC) was used in order to measure the risk of the potential attacks for the attack scenarios. In each of the cases, the TTC was computed from the beginning of the attack until the attack succeeded with the final attack step. That is, for each attack scenario, the beginning of the attack is when the attacker first enters the MSA by performing the first attack step toward either the API Gateway, in case of the API Gateway pattern, or toward the control plane, in case of the sidecar pattern. The end of the attack is when the attacker reaches the final goal, in other words, succeeds with the final attack step of a given attack scenario. The TTC for each attack scenario is presented in Table 8.1. A higher value for the TTC indicates a lower risk for the specific attack in the design pattern.

Attack Scenario	API Gateway pattern TTC (days)	Sidecar pattern TTC (days)
East-west traffic eavesdrop	19	35
Secret compromise	58	85
Broken access control	21	78

Table 8.1: Time to compromise (TTC) of the attack scenarios in respective design pattern

For the attack scenario from the communication attacks, as shown in Table 8.1 eavesdropping on the east-west traffic between the microservices resulted in 19 days for the time to compromise within the API Gateway pattern in comparison to 35 days for the sidecar pattern. In this case, the TTC value for the sidecar pattern is roughly $\frac{35}{19} \approx 1.84$ higher compared to the TTC value for the API Gateway pattern. This result indicates that the sidecar pattern is exposed to less risk than the API Gateway pattern regarding eavesdropping on the east-west communication according to the results from the attack simulation.

Regarding the attack scenario of compromising the secrets within a container the TTC value for the API Gateway pattern resulted in 58 days and the TTC for the Sidecar pattern resulted in 85 days indicating the sidecar pattern is exposed to less risk compared to the API Gateway pattern. Comparing the values from the attack simulations the time to compromise being $\frac{85}{58} \approx 1.47$ longer for the sidecar pattern compared to the API Gateway pattern in the case of secret compromise.

In the attack scenario for attacks crafted towards the individual microservices, the scenario of broken access control yielded a TTC of 21 days for the API Gateway pattern and a TTC of 78 days for the sidecar pattern in the attack simulations for the attack to be able to extract the data of the authentication database. The result, in this case, indicates that the sidecar pattern is exposed to less risk in comparison with the API Gateway pattern according to the attack simulations where the TTC value was $\frac{78}{21} \approx 3.71$ times greater in the sidecar pattern.

In all of the layers tested, the attack simulations showed that the MSA utilizing the sidecar pattern was more resistant to penetration than the MSA utilizing the API Gateway pattern. The attack simulations showed the biggest difference between the design patterns when comparing attacks towards a specific microservice.

8.2 Attack Graph Analysis

For each of the three layers, attack graphs were generated for the API Gateway pattern and the sidecar pattern resulting in a total of six attack graphs. In the attack graphs, the thickness of an edge indicates the number of attack paths passing through that specific edge where the thicker edge is the more attack paths are passing through that edge. The color of the edge represents the effort of the attacker in order to succeed with the attack step. Nodes with explanation marks attached indicate that the attack step contains defenses that can be set. The analysis of the attack graphs is divided by the layers of the MSA, that is in Section 8.2.1 an analysis of the east-west traffic eavesdrop is given, in Section 8.2.2 an analysis of compromising secrets of a Docker container is given and lastly, in Section 8.2.3 an analysis of extracting data from the database is given.

8.2.1 East-west traffic eavesdrop

The attack graph for eavesdropping on the east-west traffic is visualized in Figure 8.1 for the API Gateway pattern and in Figure 8.2 for the sidecar pattern. In both cases, the attacker needs to be able to access the east-west traffic either via the chat service sub-network or the auth service sub-network.

In the API Gateway pattern, the entry point for the attack is the API Gateway where the attacker can continue either towards the microservice for authentication or the microservice for chat functionality. Independent of the attack path chosen at this point it will lead to a new decision, whether the attacker should target the Docker container or the REST API of the microservice. Ideally, the only ports exposed by the Docker container would be the port for communicating with the REST API, consequently limiting the attacker to continue along the branch targeting the Docker container. On the other hand, if the attacker is able to interact with the Docker container further attacks can be carried out in order to gain full access to the container.

The alternative paths for the attacker are to target the REST APIs. In this case, the REST API must be exposed to the network in order for the API Gateway to be able to route requests to it. Hence the attack surface at this stage is similar to web servers or web frameworks where the attacker is trying to find vulnerabilities to exploit within the application. Thus the outcome of whether the attacker succeeds at this stage or not is dependent on the implementation by the developers.

In both cases, the attacker must be able to gain full access to the asset

in order to continue to the next attack step of accessing the network and connections to ultimately be able to access the east-west traffic.

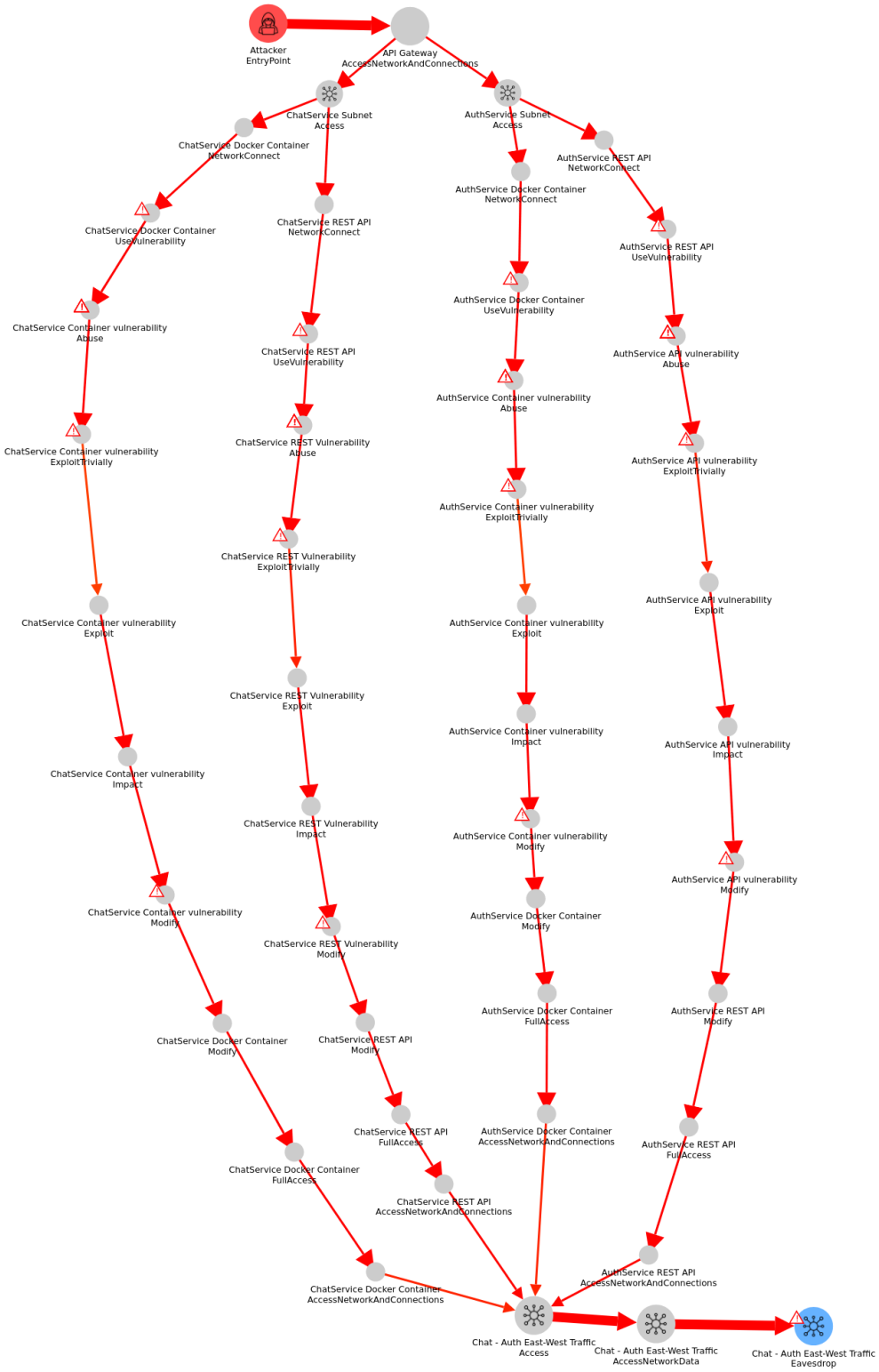

Figure 8.1: Attack graph for eavesdroping on east-west traffic in API Gateway pattern

In the sidecar pattern, the entry point of the attacker is through the control plane. Similar to the API Gateway pattern the following attack step can either be to ChatService sub-network or the AuthService sub-network. In both cases, the following attack steps are equivalent where the attacker targets the sidecar proxy of the microservice. This is because the sidecar proxy controls the east-west traffic between the microservices as it is a part of the data plane. Consequently, the attack graphs show that the attacks will be geared towards the infrastructure of the architecture rather than individual microservices' REST APIs. This is a deviation from the result which distinguishes the attack paths from the API Gateway pattern where the attacker is able to target the REST APIs. Instead, the attack steps consist of using vulnerabilities to exploit the sidecar proxy eventually leading to the attacker gaining full access of the sidecar. As full access has been reached the attacker can continue towards accessing the east-west traffic and eventually perform the eavesdrop.

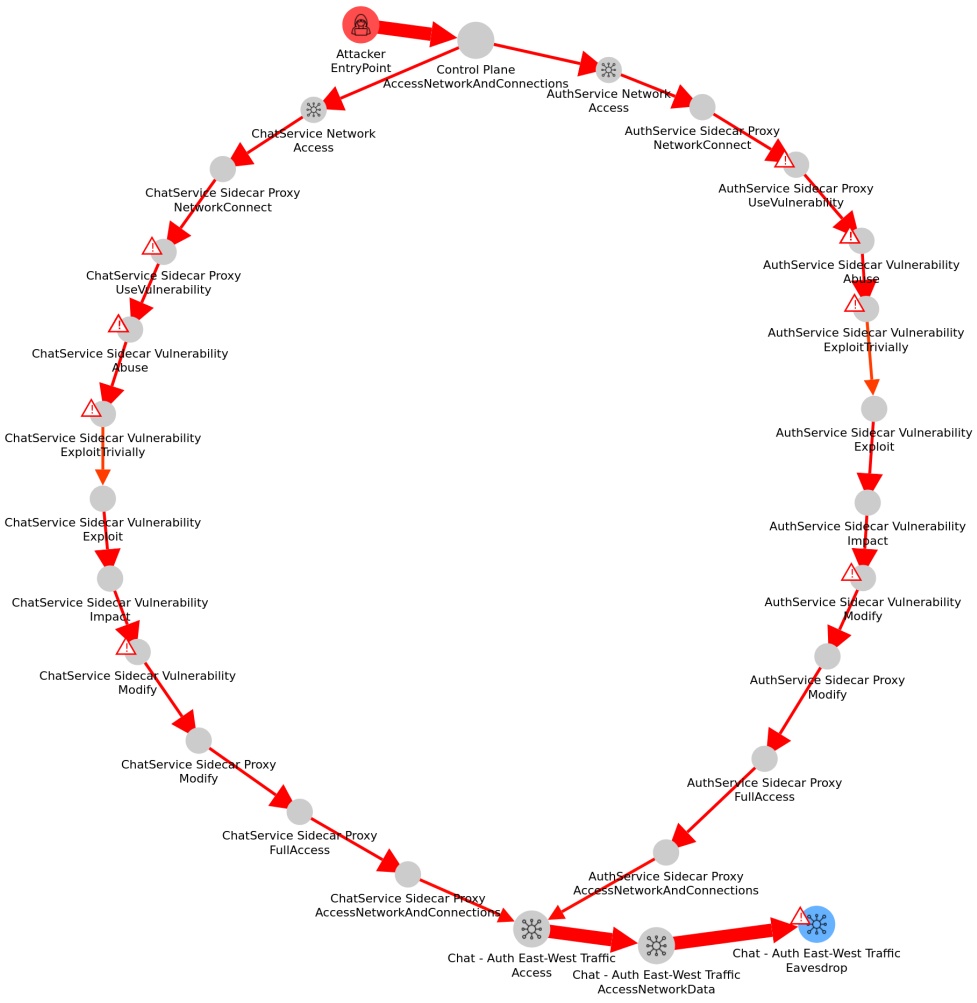

Figure 8.2: Attack graph for eavesdrop on east-west traffic in Sidecar pattern

8.2.2 Secret Compromise

Attacking the virtualization by compromising secrets from the container running the microservice is visualized by the attack graph in Figure 8.3 for the API Gateway pattern and in Figure 8.4 for the sidecar pattern. The attack graphs for both of the design patterns show that all attack paths contain the AuthService Subnet Access attack step. However it would theoretically be possible to penetrate through the ChatService sub-network but this is deemed to be unlikely by the simulation, thus no such attack path is included in the attack graph.

For the API Gateway pattern, the first attack step is to access the network

and connections of the API Gateway and further continue to access the sub-network for the authentication service. As the attacker has reached this attack step, the attack graph diverges into two separate branches. In the leftmost branch, the attacker continues to the attack step of opening a network to the Docker container. If the attacker is able to interact with the Docker container the next step is to use and later exploit a vulnerability in the Docker container in order to get read access to the image resources. It should be noted that if secrets are stored within the image where the attacker has read access, there is no requirement for the attacker to escalate privilege. However, storing secrets in plain text within the image can be considered bad practice.

In the rightmost branch, the attacker performs the attack step AuthService Docker Container AttemptReverseReach. That is, the attacker is attempting to manipulate the Docker container to open a connection back to the attacker. This could for example be if the attacker successfully performs a remote code execution by writing to the ”RUN” command in the Dockerfile attached to the image, consequently manipulating the Docker container to establish a connection to a command and control server possessed by the attacker. If the attacker succeeds at this stage the next step is to extract the secrets to reach the goal.

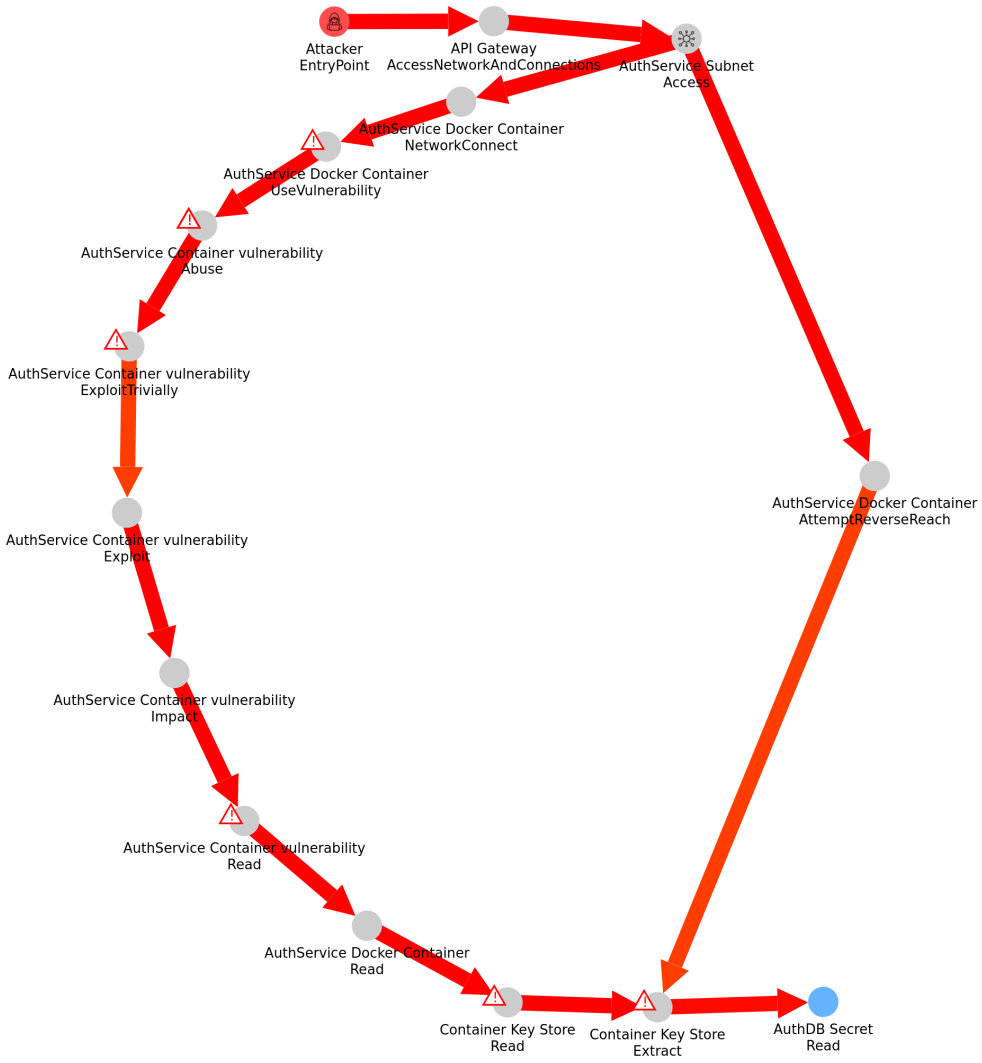

Figure 8.3: Attack graph for compromising container secret in the API gateway pattern

In the sidecar pattern, the attacker will similar to the API Gateway pattern have the possibility to choose two different branches if the AuthService Network Access is reached. However, both branches contain a greater number of attack steps in comparison with the API Gateway pattern.

If the leftmost branch is chosen, the first step is to establish a network connection to the sidecar. The attacker will then need to propagate from the sidecar to the Docker container. In order to accomplish this, the attack graph shows that full access to the sidecar proxy is required. Hence the

attacker must break the access control of the sidecar proxy. This is interesting because authorization is one of the responsibilities of the sidecar. If this step is successfully reached, however, the attacker will be able to interact with the Docker container. From this stage, the rest of the attack path to reading the secret is equivalent to the attack paths from the API Gateway pattern.

The right-most branch consists of two attack steps before the branches merge. Similar to the API Gateway pattern this path consists of manipulating the Docker container to establish a connection back to the attacker. But in contrast to the API Gateway pattern, the attack graph for the sidecar pattern shows that the attacker will attempt a reverse reach of the sidecar proxy first. This is expected as all the traffic reaching the AuthService Container will pass through the sidecar proxy. However, depending on the implementation of the control plane, the sidecar proxy might have rules set to block the AuthService Docker Container to establish a connection back to the attacker but it might be possible to establish a connection to the sidecar. But whether this is possible or not is highly dependent on the implementation. Thus it is hard to draw any conclusion regarding this path for the general case.

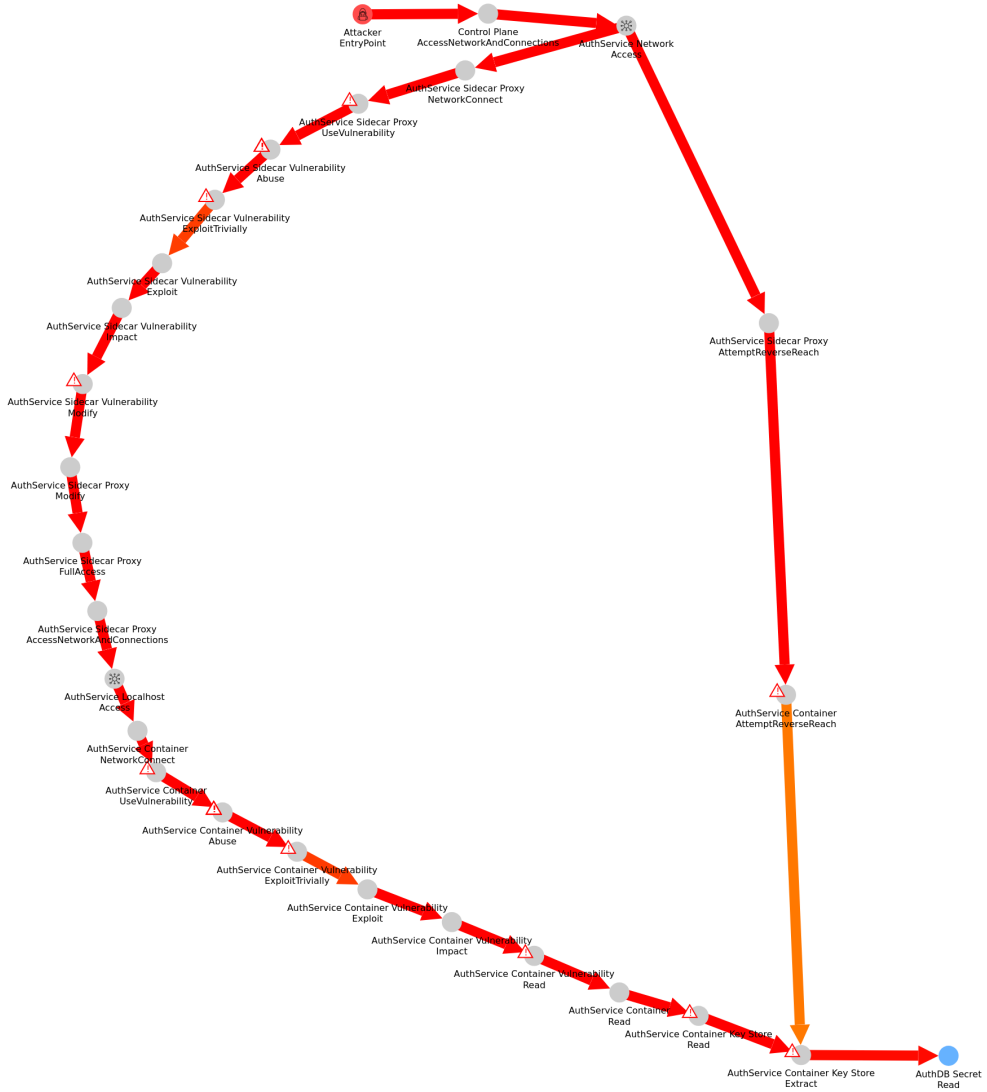

Figure 8.4: Attack graph for compromising container secret in the sidecar pattern

8.2.3 Broken Access Control

The attack graphs for the scenario of extracting data from the AuthDB can be found in Figure 8.5 for the API Gateway pattern and in Figure 8.6 for the sidecar pattern.

In the API Gateway pattern, three possible branches emerge if the attacker can successfully access the sub-network for the authentication service.

The leftmost branch consists of attempting to reverse reach the Docker container. If the attacker succeeds at this stage the attacker immediately reaches the goal to extract the data from the AuthDB according to the attack graph. However, in reality, it is likely that there are authentication mechanisms implemented in the database management system that the attacker is required to bypass in order to extract the data.

In the middle branch, the attacker attempts a reverse reach on the REST API. This could for example occur if the application is vulnerable to Server-Side Request Forgery (SSRF) and the attacker provides a URL with malicious payload to instantiate a reverse shell that the AuthService request. However, similar to the leftmost branch it is reasonable to assume that additional attack steps would be needed in order to extract the data from the AuthDB.

In the rightmost branch, the attacker establishes a connection to the REST API in contrast to the other branches where the attacker manipulates the service to establish a connection back to the attacker. If the REST API contains vulnerabilities the attacker will try to use and exploit them in order to grant read access to the database. The attacks performed at this stage could for example be SQL injections or force browsing endpoints lacking proper access control. However, which attacks are performed or whether the application is vulnerable at all is highly dependent on the implementation.

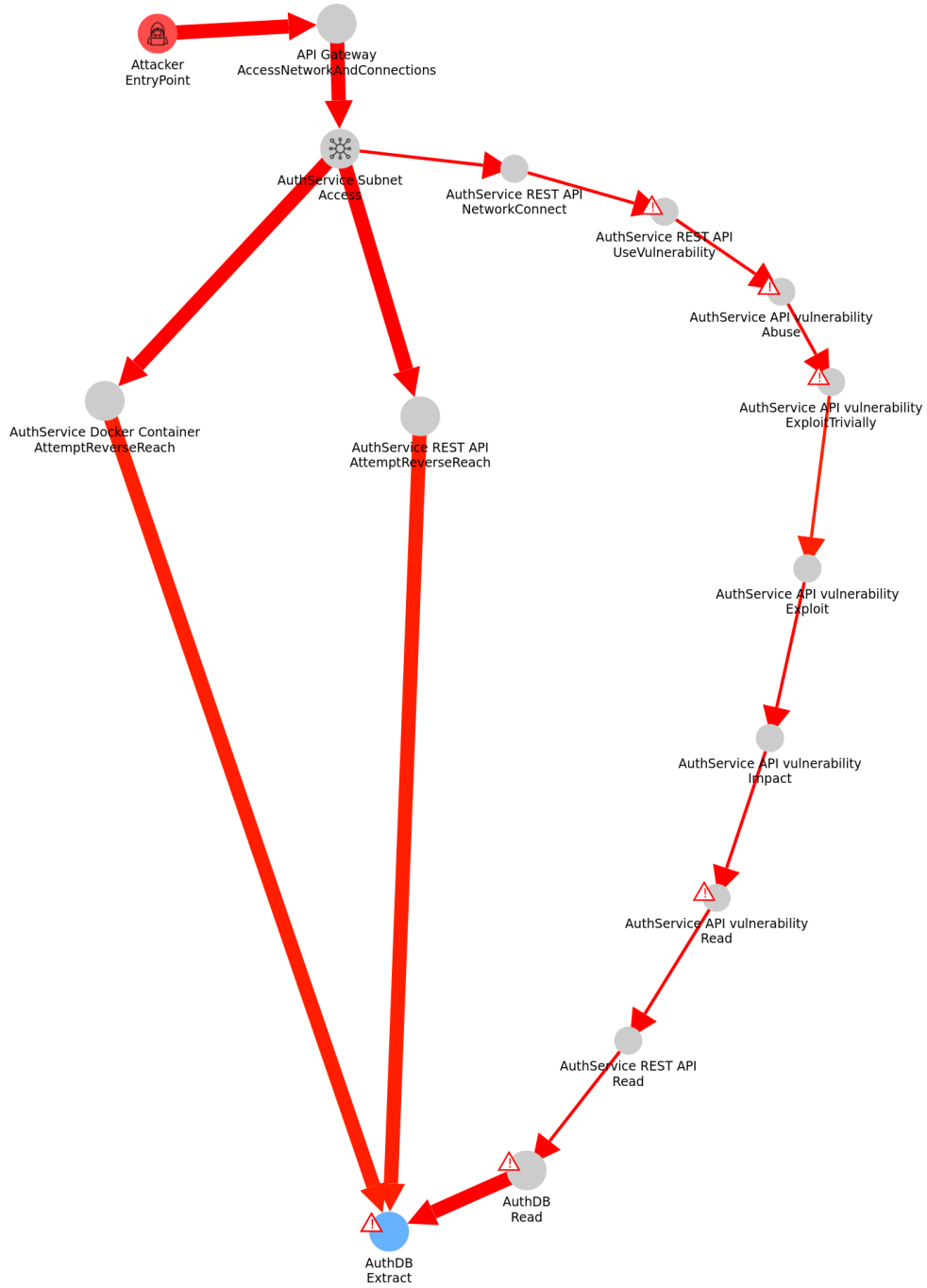

Figure 8.5: Attack graph for extracting data of the database in the API gateway pattern

In the sidecar pattern, after the AuthService Network Access attack steps

the attack graph diverges into two branches. If the attacker chooses the leftmost branch 30 additional attack steps are required in order to reach the goal. The attack graphs show in this case that the attacker first will have to connect to the sidecar of the AuthService and exploit a vulnerability to gain full access to the sidecar. This same procedure will then have to be repeated for the sidecar attached to the AuthDB. The fact that the procedure will be repeated does not necessarily significantly more effort required by the attacker in this case, as the vulnerability in the sidecar can be the same.

If the attacker gains full access to the sidecar attached to the AuthDB, the next attack step is to access the localhost and start to interact with the Docker container. The attacker will then have to find and exploit a vulnerability within the Docker container which grants the attacker read access to the database.

The attack path for the rightmost branch shares similarities with the rightmost branch of the attack graph for compromising secrets in the sidecar pattern. The difference here is that an extra attack step is needed to reach the goal. However as shown in the attack graph, the attacker will first attempt to reverse reach the sidecar proxy attached to the AuthService. Given that the attack step is successful it is possible that a similar procedure can be repeated in order to reverse reach the sidecar attached to the AuthDB. Lastly, the attacker will attempt to reverse reach the AuthDB Docker container which is equivalent to the aforementioned scenario.

Regarding both branches, it should be noted that in the asset model it was expressed that the credentials needed in order to authenticate towards the AuthDB are stored within the AuthService Container and not in the AuthDB Container. Unfortunately, it was difficult to model the access control for the database at the level of detail required to emphasize this.

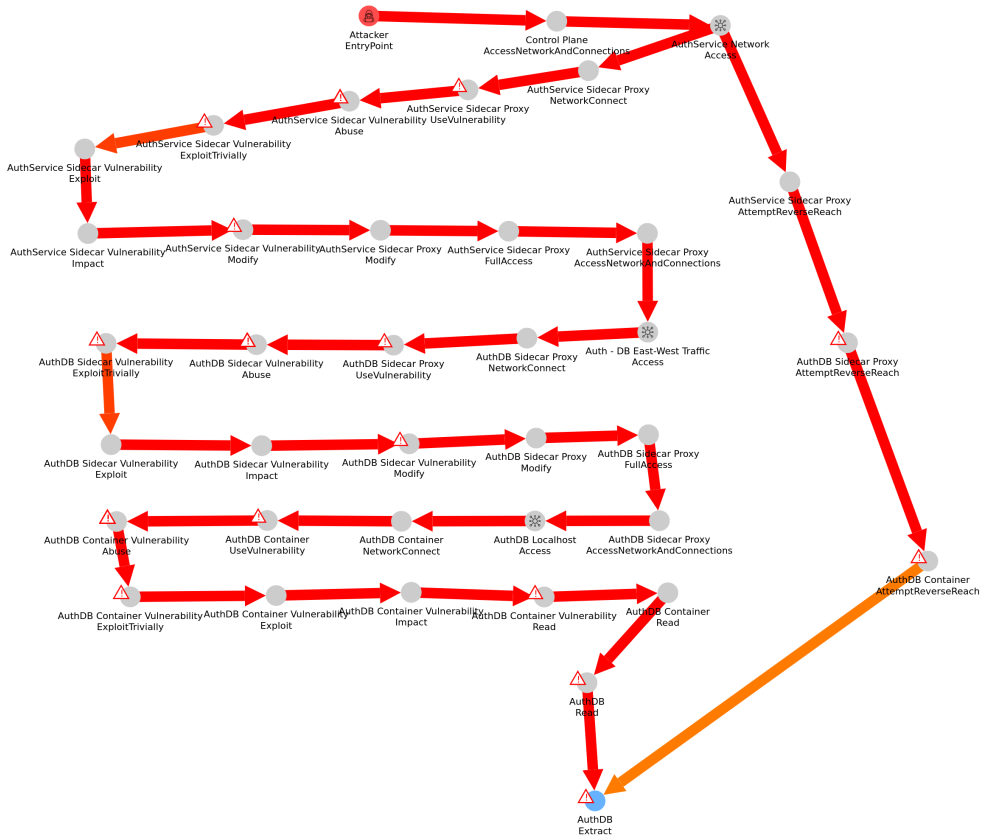

Figure 8.6: Attack graph for extracting data of the database in the sidecar pattern

8.3 Reliability Analysis

As previously described the attack simulations are dependent on probability distributions and thus not deterministic. Consequently, new attack simulations within the same environment might have slight differences in the time to compromise for the respective design patterns. Therefore in a replication of the experiment, it is possible with deviations to the results compared to the results proposed in this thesis. However, for all the attack simulations performed during this project, the result was consistent in all of the evaluated attack scenarios. Thus no such deviations from the result were shown for the experiment in this project.

8.4 Validity Analysis

The goal of this thesis is to compare the API Gateway pattern with the sidecar pattern with respect to security in order to gain insights into how the choice of design pattern will affect the security of the MSA. To reach the goal threat modeling and attack simulations were used with the software SecuriCAD in the DSL CoreLang. In order to address the validity of the result, a discussion regarding possible sources of error in the SecuriCAD implementation will be given in the following paragraph.

8.4.1 Attack graph validity

In the API Gateway pattern since the API Gateway is the entry point to the application a key responsibility it has is to prevent abuse of the application and to provide authorization and access control. However, the robustness of the API Gateway depends on the implementation which comes in a wide range of options, from commercialized options to case-specific implementations. Therefore it is hard to model a propitiate level of difficulty needed for an attacker to bypass the API Gateway for the general case. Furthermore, it is hard to model specific functionality by the API Gateway. One such example is authentication, in the modeled application a realistic scenario would be that the API Gateway allows a client to interact with the authentication service in order to sign in but blocks clients to interact with the chat service if the client has not yet been authenticated. Hence in some cases, the attacker is required to bypass the authorization and in some cases not, consequently this means extra attack steps are needed to bypass authorization which likely would imply a longer time to compromise. However, in the application under evaluation, it is assumed that anyone can sign up for an account and thus should be able to interact with the chat service.

A similar example happens for both the design patterns in the scenario of extracting data from the database. In order to obtain the data from the database, the attacker would typically need to bypass the authorization in the database management system, likely by stealing the database credentials. This would increase the number of attack steps as well. On the other hand, the authorization mechanisms and how credentials are stored are highly dependent on the implementation and therefore hard to model for the general case.

Regarding the containerization, the containers were modeled as Application assets in CoreLang. This was a design choice made at the beginning of the project which there was uncertainty about. However, I do believe that the

Application asset is sufficient in order to model the containers for the scope of this project. On the other hand in order to be able to model the containers at a finer level of granularity an extension to CoreLang with a specific asset to represent containers is needed.

8.4.2 Analysis of external sources

While no previous research comparing the risks of the design patterns have been identified, there is previous work analyzing the security challenges of the design patterns.

In [48] Jander et al. conclude that MSAs in many cases delegate the authentication and authorization to the API Gateway. As a consequence enables the possibility of an evil actor to access any of the individual services if the actor manages to exploit vulnerabilities in the API Gateway. Therefore if the API Gateway is the only component that contains the responsibility of authentication and authorization and thereby is viewed as a shield protecting microservices the MSA will have insufficient defense in depth. In the result, this is reflected in all attack graphs for the API Gateway pattern where the attacker is able to interact directly with the individual services after successfully bypassing the API Gateway in contrast to the sidecar pattern where the attacker by nature of the pattern will interact with the sidecar.

In [9] the authentication and access control are further addressed stating that the API Gateway can be used to provide both authentication and access control centralized in the MSA. Furthermore, it is emphasized that some form of central architecture for authorization is required as the MSA can contain a great number of microservices. However, if the API Gateway pattern is used, it is also suggested to implement mutual authentication in order to mitigate the risks of anonymous connections to the microservices. Regarding access policies it is suggested that the API Gateway should enforce it at a broader level such as permissions to interact with a set of functionalities whereas a more detailed level of authorization for individual microservice should be provided close to, or by the microservices themselves. For communication, the article highlights that secure communication both north-south and east-west are critical parts of the MSA and that both types of traffic should be encrypted and with the use of mutual authentication for example by mTLS. Furthermore, clients should target a single gateway URL instead of targeting the individual microservices directly. Regarding the sidecar pattern, it is emphasized that the security policies in the control plane are correctly defined to ensure that the service mesh does not introduce new vulnerabilities to the MSA. Therefore it is

suggested to enable policies for access control by default for all services and to avoid configurations that might enable the possibilities of privilege escalation. Furthermore, it is stated that there is a threat of bypassing the sidecar proxy which needs to be handled for highly sensitive MSAs.

In the modeled scenarios of the sidecar pattern, bypassing the sidecar proxy never occurred in the attack graph which might have a negative impact on the validity of the result. It is also interesting to point out that all attack paths for the sidecar pattern either contain attack steps of attempting to reverse reach the sidecar or gaining full access over the sidecar which corresponds to flawed configuration and escalation of privileges respectively.

8.5 Suggested mitigations

This section presents suggestions regarding how security risks in MSAs can be mitigated. In Section 8.5.1 suggestions regarding authentication and authorization in MSAs are discussed, Section 8.5.2 contains a discussion of how east-west traffic can be secured followed by Section 8.5.3 where suggestions regarding how API and database credentials can be stored. Lastly, in Section 8.5.4 a discussion on how security practices can be integrated into the development process is given.

8.5.1 Access control

Authentication and authorization in MSAs are important at the edge of the system but also within the system between the services. Implementing authentication and authorization at the edge level of the system, e.g. as an API Gateway, is intuitive as that is the point where trust boundaries are crossed. However, if an evil actor manages to bypass the authorization on the edge level the system needs defense in depth to mitigate the impact of the actions. Therefore it is important to implement security mechanisms for authentication and authorization at individual services as well.

For a MSA utilizing the API Gateway pattern, it becomes important as the pattern itself does not provide any functionality regarding authentication and authorization between services. In order to guarantee that a microservice is authorized to request a specific resource security JSON Web Tokens (JWT) along with security frameworks such as OAuth 2.0 can be used [49]. Even though the JWT does not authenticate the services towards each other, it can be used to authorize the end-user at the microservice to own the resource even if the request is propagated through multiple services [50]. In this way, by

always verifying the request there is no need for the services to trust each other, hence zero trust between the services can be achieved.

For a MSA utilizing the sidecar pattern, this type of functionality is already provided by the design pattern by the service mesh. However in this case it is important with proper implementation and configuration of the service mesh to ensure that policies are consistently met for all services such as enabling access control by default.

8.5.2 Communication

To mitigate risks of eavesdropping on data in transit encryption should be used both for client-to-service communication and service-to-service communication. In both cases, transport layer security (TLS) can be used to achieve this. For the client-to-service communication, this is the standard for the traffic between the user and the entry point of the application (e.g. the API Gateway) as most modern web browsers will mark web pages using HTTP instead of HTTPS as insecure. For the service-to-service communication, TLS can be combined with mutual authentication in order to prevent unauthenticated communication within the MSA by the use of mTLS. The use of mTLS would provide the functionality of authentication between the services by asymmetric cryptography with a signed certificate [9]. However, it requires an additional component, a certificate authority (CA) in which all microservices trust, but the CA can be located within the MSA and be integrated into the deployment process for the microservices [50]. Since the traffic is encrypted and signed this will mitigate an attack where an adversary is intercepting data in transit both by the fact that it is a cipher text, hence the adversary is required to bypass the encryption in order to extract the information, but also as the services can verify who the sender is and take actions accordingly.

A couple of the attack graphs contained the attack step attempt reverse reach, in order to mitigate this all communication entering and leaving the MSA should pass through a single component, such as the API Gateway, where the data is inspected and filtered. In the component, the rules for network traffic should be properly set for the given MSA. In some cases, the microservices does not have any requirement to establish outgoing connections to hosts outside the MSA while in other cases outgoing connection to for instance third-party services are needed. Therefore the rules depend on the requirements of the MSA. If the sidecar pattern is used further rules can be set for the individual services in the service mesh.

8.5.3 Containerization

Independent of which design pattern is used chances are high that the microservices will be deployed in containers. Container security is a research topic of its own but a brief discussion of the mitigations relevant to this project will be discussed in the following section.

In order to mitigate the risk of the attacker gaining full access to the container, the principle of least privilege should be applied. As the Docker container might need to install dependencies the container will be executed as *root* user if not explicitly stated otherwise. However, there are prevention mechanisms utilized by Docker, such as Linux namespaces, to limit the container to only work within its designated set of permissions [51]. If this configuration is not properly applied and an adversary manages to break out of the application into the container, as a consequence the adversary will gain root privileges within the container and therefore be able to install tools to further escalate the attack. In order to mitigate this, the container can be configured to run as a user with the only permissions required to perform its tasks. On the other hand, it was argued in [4] that this mitigation is not sufficient alone and that the containers need SELinux enabled to enforce the access control and further isolate the container process itself.

In one of the performed attack simulations, the goal was to extract secrets such as database credentials or API keys. Intuitively, managing secrets can be a difficult task in MSAs when more services are added. It might be tempting to store secrets within the container for example as environment variables. However, this is a bad practice as if the attacker manages to gain access to the container the environment variables can be listed. To mitigate this scenario, secrets can be stored outside of the container to be sent over a secure channel to the container at run time. This can be achieved for instance by binding the location of the secrets with the secret argument in Docker. As a result, the secrets will be encrypted in transit, and in the file they are stored at rest in the container [52]. However, the secrets should also be encrypted on the host machine and managed in a way that they can be rotated. There are several techniques and tools that can be utilized to manage secrets in a secure manner provided by orchestration frameworks such as Kubernetes or products by cloud service providers such as Key vault if Microsoft azure is used.

In order to mitigate the risk of an adversary exploiting misconfigurations of the Docker containers and images tools such as Docker Secure Bench can be used in combination with other static analysis tools to find known vulnerabilities.

8.5.4 Security in software engineering practices

In addition to the aforementioned suggestions general practices that can be applied to MSAs are summarized in the following section.

First and foremost, threat modeling can be used continuously during the lifetime of the system. Since one of the reasons among others for choosing an MSA as the architecture in the first place is to be able to add new services and scale the system, it can lead to a large system both in terms of the code bases and infrastructure but also the number of people involved. Integrating threat modeling in the workflow can increase the awareness of the threats to everyone involved in the development process of the system and should not be practiced by a closed-off group such as the security engineers.

In order to improve the security of any system including microservices, reducing bugs and unexpected behavior of the system is of the essence. Hence, high test coverage of the code throughout the system is important to ensure the quality of the code and mitigate the risks of an adversary taking advantage of weaknesses in the code. For the use of third-party libraries tools can be used in order to get notified of known vulnerabilities along with patched versions to ensure that the libraries used is up to date.

Chapter 9

Conclusions and Future work

In this chapter, the conclusions, limitations, future work, and reflections for the thesis are given. In Section 9.1 the conclusions are presented. Section 9.2 presents the limitations of the project. In Section 9.3 a description regarding the future work is given and lastly in Section 9.4 the reflections regarding the projects are presented.

9.1 Conclusions

In all of the evaluated attack simulations, the sidecar pattern demonstrated less risk in comparison with the API Gateway pattern with respect to TTC. The sidecar pattern contains more abstraction by nature of the pattern which may have a positive effect on the robustness of the system as the adversary is required to compromise more components in comparison with the API Gateway pattern. For the attack graphs, the sidecar pattern contained less than or an equal number of attack paths in comparison with the API Gateway pattern as all communications pass through the sidecar and the attacker was unable to interact directly with the services. As a consequence, the isolation of communication provided by the sidecar pattern appears to yield positive effects on the security. On the other hand, if an adversary is able to bypass the isolation of communication the number of attack paths in the sidecar pattern would probably increase. Thus, whether or not it is possible to bypass the sidecar needs further investigation.

The goal of this thesis was to compare the API Gateway pattern with the sidecar pattern with respect to security in order to gain insights into how the choice of design pattern will affect the security of the MSA and provide knowledge of what actions can be taken to increase the security of a MSA.

An overview of potential threats to MSAs was given in the literature study. While the design patterns could not be compared for all possible threats great insights were made regarding the abstraction provided by the sidecar pattern and the essence of properly implemented access control, especially for the API Gateway pattern. A presentation of suggested mitigations was provided but further investigation is needed to fully evaluate the presented actions.

For the MSA evaluated in this project the assets and attack steps provided by CoreLang along with the functionality provided by SecuriCAD were sufficient in order to model the MSA. In more detailed models of architectures utilizing specific technologies, extensions to CoreLang or a specific DSL might be needed in order to express complex scenarios related to access control and containerization.

9.2 Limitations

While the three attack scenarios evaluated in this project provided insights regarding the security both for the specific design patterns and for MSAs as a whole, not all possible attacks were evaluated. This was primarily due to two reasons. Firstly it was not possible for the scope of the project to evaluate all identified threats. Secondly, some of the potential attacks were fairly specific to certain technologies, especially the attacks related to containerization, as a consequence, it was uncertain whether the attack simulations for those attacks would yield a valid result using CoreLang alone.

Since the method used in the project for threat modeling is by the use of attack graphs, it was described previously that there is a possibility that attack paths are excluded from the attack graphs. A limitation of this work is that it is hard to know whether the attack graphs contain all relevant attack paths and if prominent attacks provided by previous research are covered by the attack graphs or not. Furthermore, new types of attacks and defenses are invented and not taken into account by the attack graphs in this thesis.

One of the problems encountered during this project was choosing a proper probability distribution regarding the local time to compromise value of the assets. In some cases, the probability distribution can be chosen with some degree of confidence as it has been extensively tested in previous research, such as the time it takes to crack a password for instance. In other cases, it requires a deeper knowledge of specific technologies or more assumptions and is therefore harder to accurately choose.

9.3 Future work

In addition to the layers investigated in this thesis, MSAs also face threats related to hardware, cloud environment , and orchestration which were for the scope of this thesis not investigated further. As MSAs are commonly deployed on hardware owned and managed by cloud service providers it is difficult to make any informed decisions regarding the threats and countermeasures and was therefore left out of this project. The cloud environment and orchestration on the other hand are managed by the organization owning the MSA and therefore have the possibility to take action regarding the threats and countermeasures. Even though cloud environment and orchestration are not specific to MSAs per se, further investigation regarding the threats they face is needed in order to fully identify all threats to a MSA as they are heavily used. Regarding cloud environments, future work could utilize existing domain-specific languages for the cloud built upon MAL, such as AzureLang or AWSLang. For the orchestration, on the other hand, future work could include the construction of a new DSL for the threats related to the orchestration.

Regarding the evaluated layers, future work could include more attack simulations or different attack scenarios to further compare the design patterns investigated in this thesis. Investigating all of the identified threats for this thesis would provide completeness for the comparison of the design pattern. Furthermore, it could also be used in order to strengthen or weaken the result of this thesis depending on if the results are contrary or unanimous with this thesis.

Since the thesis uses attack simulation another aspect future work could investigate is the validity of the attack simulations. That is, how well the attack simulations performed in this project correspond with reality. This could be accomplished by implementing or using a MSA in production to perform penetration tests against, based on the attack paths provided by this project. As a consequence further insights could be gained regarding the risks and possibly new attack paths not presented in this project could be discovered.

Finally, regarding the suggested mitigations further investigation is needed in order to evaluate what effect they have on the security of the system. Hence future work for instance evaluates a MSA utilizing the API Gateway pattern with and without implemented defense in-depth mechanisms in order to conclude whether or not it increases the security of the system.

9.4 Reflections

Being aware of the threats and how the security can be increased is important to any system involving users. As the digitization continues the population becomes more dependent on the technology, it is therefore of essence to ensure its integrity both for the sake of the users and in the interest of the stakeholders. While this project did not investigate any microservice architectures in production, they are commonly implemented with scalability in mind. The threats identified by this thesis along with the suggested mitigations can increase the security of the system and thereby the users before any violations occurred and could therefore be used in order to target United Nations Sustainable Development Goals (SDGs) goal number 9: ”Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation” [53].

The ethical aspect of this thesis mainly concerns how the result is used. While the project does not evaluate any MSA in production, it provides insights into the security challenges within MSAs. The attack graphs provide information regarding what attack steps are necessary in order to compromise the system. Even though a different MSA likely would differ in some aspects compared to the evaluated MSA, the information obtained from the attack graphs could have consequences not intended by this project if it is used with malicious intent.

For organizations and developers working with MSAs this thesis can act as a basis to gain insights into the security challenges associated with the investigated design patterns along with suggestions for existing techniques to mitigate these challenges. The methodology used in this project can further be applied to MSAs with similar structures or modified in order to model a different MSA.

References

[1] L. Miller, P. Mérindol, A. Gallais, and C. Pelsser, "Towards secure and leak-free workflows using microservice isolation," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, 2021. doi: 10.1109/HPSR52026.2021.9481820 pp. 1–5. [Page 5.]

[2] N. Shevchenko, T. A. Chick, P. O'Riordan, T. P. Scanlon, and C. Woody, "Threat modeling: A summary of available methods," 2018. [Page 6.]

[3] P. Johnson, A. Vernotte, M. Ekstedt, and R. Lagerstrom, "pwnpr3d: An attack-graph-driven probabilistic threat-modeling approach," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2016. ISBN 1509009906 pp. 278–283. [Page 6.]

[4] D. Yu, Y. Jin, Y. Zhang, and X. Zheng, "A survey on security issues in services communication of microservices-enabled fog applications," *Concurrency and computation*, vol. 31, no. 22, p. n/a, 2019. [Pages 7 and 66.]

[5] P. Nkomo and M. Coetzee, "Software development activities for secure microservices," in *Computational Science and Its Applications – ICCSA 2019*, S. Misra, O. Gervasi, B. Murgante, E. Stankova, V. Korkhov, C. Torre, A. M. A. Rocha, D. Taniar, B. O. Apduhan, and E. Tarantino, Eds. Cham: Springer International Publishing, 2019. ISBN 978-3-030-24308-1 pp. 573–585. [Page 8.]

[6] "Microservices." [Online]. Available: https://martinfowler.com/articles/microservices.html [Page 10.]

[7] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017,

pp. 195–216. ISBN 978-3-319-67425-4. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12 [Pages 10 and 11.]

[8] "Microservices security cheat sheet." [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Microservices_security.html [Pages 11 and 12.]

[9] R. Chandramouli, "Microservices-based application systems," *NIST Special Publication*, vol. 800, no. 204, pp. 800–204, 2019. [Pages 11, 12, 13, 14, 28, 63, and 65.]

[10] S. M. Jain, *Linux Containers and Virtualization: A Kernel Perspective*. Berkeley, CA: Apress L. P, 2020. ISBN 9781484262825 [Page 11.]

[11] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *Journal of network and computer applications*, vol. 119, pp. 97–109, 2018. [Page 11.]

[12] F. Montesi and J. Weber, "Circuit breakers, discovery, and api gateways in microservices," 2016. [Page 12.]

[13] S. Gadge and V. Kotwani, "Microservice architecture: Api gateway considerations," *GlobalLogic Organisations, Aug-2017*, 2018. [Page 13.]

[14] Microsoft, "Sidecar pattern," Dec 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar [Page 14.]

[15] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: an industrial survey of microservice tracing and analysis," *Empirical software engineering : an international journal*, vol. 27, no. 1, pp. 25–25, 2021. [Page 14.]

[16] G. Miranda, *The Service Mesh*. O'Reilly Media, Inc, 2018. ISBN 9781492031321 [Page 14.]

[17] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 122–1225. [Page 14.]

[18] A. Shostack, *Threat modeling : designing for security*, 1st ed., 2014. ISBN 1-118-82269-2 [Pages 16, 17, and 18.]

[19] B. Schneier, "Attack trees," *Dr. Dobb's journal*, vol. 24, no. 12, pp. 21–29, 1999. [Pages xi, 17, and 18.]

[20] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 workshop on New security paradigms*, 1998, pp. 71–79. [Page 19.]

[21] P. Johnson, R. Lagerström, and M. Ekstedt, "A meta language for threat modeling and attack simulations," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018. New York, NY, USA: Association for Computing Machinery, 2018. doi: 10.1145/3230833.3232799. ISBN 9781450364485. [Online]. Available: https://doi.org/10.1145/3230833.3232799 [Page 20.]

[22] S. Katsikeas, S. Hacks, P. Johnson, M. Ekstedt, R. Lagerström, J. Jacobsson, M. Wällstedt, and P. Eliasson, "An attack simulation language for the it domain," in *Graphical Models for Security*, H. Eades III and O. Gadyatskaya, Eds. Cham: Springer International Publishing, 2020. ISBN 978-3-030-62230-5 pp. 67–86. [Page 21.]

[23] Mal-Lang, "corelang.mal at master · mal-lang/corelang," Mar 2021. [Online]. Available: https://github.com/mal-lang/coreLang/blob/master/src/main/mal/coreLang.mal [Page 21.]

[24] "securicad enterprise," 2021. [Online]. Available: https://foreseeti.com/securicad-enterprise/ [Page 24.]

[25] T. Yarygina and A. H. Bagge, "Overcoming security challenges in microservice architectures," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2018. doi: 10.1109/SOSE.2018.00011 pp. 11–20. [Pages 26 and 30.]

[26] "Network denial of service." [Online]. Available: https://attack.mitre.org/techniques/T1498/ [Page 27.]

[27] "Adversary-in-the-middle." [Online]. Available: https://attack.mitre.org/techniques/T1557/ [Page 27.]

[28] V. Ramachandran and S. Nandi, "Detecting arp spoofing: An active technique," in *Information Systems Security*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. ISBN 3540307060. ISSN 0302-9743 pp. 239–250. [Page 27.]

[29] “Capec-151 identity spoofing,” Oct 2021. [Online]. Available: https://capec.mitre.org/data/definitions/151.html [Page 27.]

[30] “Eavesdrop on insecure network communication.” [Online]. Available: https://attack.mitre.org/techniques/T1439/ [Page 27.]

[31] S. Malladi, J. Alves-Foss, and R. B. Heckendorn, *On Preventing Replay Attacks on Security Protocols*, 2002. [Page 27.]

[32] “The heartbleed bug in openssl library,” Apr 2014. [Online]. Available: http://heartbleed.com/ [Page 27.]

[33] B. Möller, T. Duong, and K. Kotowicz, “This poodle bites: exploiting the ssl 3.0 fallback,” *Security Advisory*, vol. 21, pp. 34–58, 2014. [Page 27.]

[34] Y. Sheffer, R. Holz, and P. Saint-Andre, “Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS),” RFC 7457, Feb. 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7457 [Page 27.]

[35] “A01:2021 – broken access control,” 2021. [Online]. Available: https://owasp.org/Top10/A01_2021-Broken_Access_Control/ [Page 28.]

[36] “A02:2021 – cryptographic failures,” 2021. [Online]. Available: https://owasp.org/Top10/A02_2021-Cryptographic_Failures/ [Page 28.]

[37] “A03:2021 – injection,” 2021. [Online]. Available: https://owasp.org/Top10/A03_2021-Injection/ [Page 28.]

[38] “A04:2021 – insecure design,” 2021. [Online]. Available: https://owasp.org/Top10/A04_2021-Insecure_Design/ [Page 28.]

[39] “A06:2021 – vulnerable and outdated components,” 2021. [Online]. Available: https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/ [Page 29.]

[40] “A07:2021 – identification and authentication failures,” 2021. [Online]. Available: https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/ [Page 29.]

[41] “A08:2021 – software and data integrity failures,” 2021. [Online]. Available: https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/ [Page 29.]

[42] "A09:2021 – security logging and monitoring failures," 2021. [Online]. Available: https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/ [Page 29.]

[43] "A10:2021 – server-side request forgery," 2021. [Online]. Available: https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_(SSRF)/ [Page 29.]

[44] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE access*, vol. 7, pp. 52 976–52 996, 2019. [Pages 30 and 31.]

[45] R. Weilin Zhong, "Code injection," 2021. [Online]. Available: https://owasp.org/www-community/attacks/Code_Injection [Page 30.]

[46] L. Rice, *Container security : fundamental technology concepts that protect containerized applications*, 1st ed., 2020. ISBN 1-4920-5669-3 [Pages 30 and 31.]

[47] "Cve-2019-5736," 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-5736 [Page 31.]

[48] K. Jander, L. Braubach, and A. Pokahr, "Defense-in-depth and role authentication for microservice systems," *Procedia computer science*, vol. 130, pp. 456–463, 2018. [Page 63.]

[49] D. Sawano, D. Bergh Johnsson, and D. Deogun, *Secure by Design*. New York: Manning Publications Co. LLC, 2019. ISBN 9781617294358 [Page 64.]

[50] W. K. A. N. Dias and P. Siriwardena, *Microservices Security in Action*. New York: Manning Publications Co. LLC, 2020. ISBN 1617295957 [Pages 64 and 65.]

[51] "Isolate containers with a user namespace," https://docs.docker.com/engine/security/userns-remap/, accessed: 2022-05-25. [Page 66.]

[52] "Manage sensitive data with docker secrets," https://docs.docker.com/engine/swarm/secrets/, accessed: 2022-05-25. [Page 66.]

[53] "United nations - sustainable development goals: Goal 9," https://sdgs.un.org/goals/goal9, accessed: 2022-05-25. [Page 72.]

TRITA-EECS-EX-2022:610