FACULTY OF ENGINEERING AND TECHNOLOGY
MASTER OF SOFTWARE ENGINEERING (SWEN)
MASTER THESIS

An Authentication/Authorization Approach for a Microservices Architecture

---

Author:
Mohammad Arouri
(1165409)

Supervisor:
Dr. Yousef Hassoneh

August 18, 2020

جامعة بيرزيت
BIRZEIT UNIVERSITY

An Authentication/Authorization Approach for a Microservices Architecture

---

Author

Mohammad Arouri

This thesis was prepared under the supervision of Dr. Yousef Hassouneh and has been approved by all members of the examination committee

Dr. Yousef Hassouneh, Birzeit University
(Chairman of the Committee)

Dr. Ahmad Alsadeh, Birzeit University
(Member)

Dr. Ahmed Tamrawi, Birzeit University
(Member)

Date of Defense:
August 18, 2020

Abstract

Microservices architecture is an evolving trend in software engineering that enables building large scale, highly scalable, available and flexible systems. However, microservices are not a silver bullet, they have their challenges and complexities. One of these main challenges is security.

State-of-the-art shows that microservices security and their aspects are an important challenge that is not well researched and needs more attention. Among these aspects are authentication and authorization. For microservices applications to be secure, a proper authentication and fine-grained authorization framework should be in place.

In this research, we propose a new security framework for authentication and fine-grained authorization (MSFAA) that relies on the use and coordination of a set of security standards and frameworks to tackle the security requirements in a microservices architecture. Our solution is based on a combination of OAuth2, JWT and Open Policy Agent (OPA). To evaluate our results, we adopted an industrial motivating use case, the Applicant Tracking System (ATS). On top of it, we implemented our security framework and evaluated the effectiveness of the proposed framework. To study the performance implications of our security framework, we designed and conducted an experiment in which we measured the overhead caused by the proposed security framework in terms of API latency. Our results show that the performance overhead of the

security framework is around 12%. We believe that this is an acceptable overhead due to two main reasons. The first reason is that security is an essential and critical aspect in a microservices systems. The second reason is that microservices are tolerant to API latency due to their distributed nature.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we give an overview about microservices, their definition and characteristics, compare microservices with the previous monolithic applications, explain the main advantages and disadvantages of using microservices architecture, and focus on the lack of research in microservices aspects, especially the security aspect that brought our attention. Finally, we present our research objectives, contributions and provide an overview of our next steps.

1.1 Microservices Overview

Microservices architecture is a trending architectural style in the industry, it has been gaining a lot of attention and momentum for the past few years. Big companies like Netflix [41], Amazon [4] and Uber [13] have shifted their systems to use microservices architecture. The International Data Corporation has predicted that 80% of application development on cloud platforms will be built using Microservices architecture by the end of 2021 [31]. This momentum is pushed by a set of factors; one of these factors is the enterprise movement to escape from the monolith hell. Another factor is that companies are in a continuous pursuit to modernize their existing applications [47]. Young companies and

startups use microservices seeking the joy benefits of technology heterogeneity, use the right tool for the right job, resilience, scaling, ease of deployment, better organizational alignment and easier replaceability [40].

1.2 Microservices Definition

One of the earliest definitions for microservices was provided by Lewis and Fowler [32]. They stated that microservices architecture is a development approach in which a single application is built as a group of small services, each of these services run in its own independent process. These small services communicate with each other using a lightweight protocol, usually HTTP APIs. These services are built and grouped as a reflection of business capabilities. Microservices should be independently managed, deployed and usually built using different programming languages, frameworks and storage technologies.

In his famous book "Building Microservices" [40], Sam Newman defined microservices as a small, focused, autonomous services that are built around business entities. Microservices follow important familiar concepts. One example is the "Single Responsibility Principle", which encourages to gather the functionalities that change for the same reason together and separate the other functionalities that change separately for different reasons.

Another definition for Microservices is an architectural style derived from the service oriented architecture and enriched with other principles and good practices like the unix principle 'Do one thing and do it well (DOTADIW)' [62].

1.3 Basic Microservices Characteristics

By definition, all microservices share a set of characteristics; isolation, standard interface, autonomy and fine grained. Each microservice is a standalone service that can be managed, deployed, maintained and even destroyed in isolation from other microservices. Each one of them has a standard way to communicate with other microservices as well as with the external world. Same can be said about the APIs exposed by these services, either for another microservice within the same system or for an external actor, all of these exposed APIs share a common standard way of communication.

The autonomy of the microservice is established by the idea of the independency of deployment, scaling, management and life cycle. Each microservice works in its bounded context. This basically means that each microservice does only one thing and does it well. This will make each microservice a fine grained component in its own.

1.4 Comparing Microservices with Monolithic Applications

Service oriented architecture has been used for over a decade. Microservice architecture can be considered as a new enhanced derivation of the old service oriented architecture. While service oriented architecture focuses on providing a small number of interconnected components that are usually large and complex, which are referred to as monolithic applications. Microservices tend to provide many small components that are lighter and simpler. This main practice ensures better maintainability, faster development cycles and shorter time to release [16].

Monolithic applications have a lot of disadvantages that drove developers away from using them. One of these disadvantages is code maintainability. In a monolithic application, codebases are shared between a big team of developers. If a developer wants to introduce a new concept, restructure an existing component or integrate with a new library, she needs to take approval from all involved parties. She also needs to make sure not to break anything within this large monolithic application. This makes it a hard task to accomplish and very time consuming.

Scalability is another issue of monolithic applications. If a component has more demand, the whole application needs to be scaled in order to meet this new demand. More scale means more resources, which in turn means additional cost. This makes monolithic applications scaling a pain point and a delicate decision to make.

Handling deployments is another issue in monolithic applications. If a component is modified and ready to be shipped, or if a critical bug is fixed, the entire application needs to be deployed at once [1].

Code understandability is another concern in monolithic applications. The codebase is usually large. The developer needs a lot of time to grasp the different aspects of the application structure, modularity, components and functionality.

1.5 Microservices Advantages

Microservices have a lot of benefits, including the enablement of technology heterogeneity, resilience, scaling, ease of deployment, better organizational alignment and easier replaceability [40].

When the system is decomposed into many services. Developers can decide which technology they want to use, which programming language, framework, datastore or even operating system they prefer. This will make it easier to pick the right tool for the right job.

The use of microservices enables faster technology adoption and more rapid changes; if an emerging technology appears, developers can easily use this technology in some parts of the system by applying it to a subset of microservices, without the need to apply it to the entire system at once. The capability to apply changes to a specific set of microservices without putting the entire system at risk gives the developers more confidence in applying changes rapidly and efficiently to the system.

Resiliency is a key concept in software engineering. If one component of the system fails, the overall system should not fail by not allowing the failure to cascade. Prior to microservices, this was a difficult job to achieve, due to the high coupling between the different components of the monolithic application or between the different services in a Service Oriented Architecture (SOA) based system. The principles of service boundaries and bounded context of microservies enabled them to work independently. If one of the microservices went down, that particular functionality is disrupted and the remaining parts of the system continue to work [40].

Another win for the microservices is the scale. As discussed before, scaling a monolithic application is a pain point and delicate decision to make due to the large consumption of resources. On the other hand, microservices scale is easier and more cost efficient. If a microservice is needed to scale, that particular part of the system can be scaled independently from the other parts. Microservices also

made adapting modern concepts like on-demand provisioning easier. Which opened the door for more optimal and efficient scaling, yielding to more and more cost savings [47].

Another key benefit of microservices is the ease of deployment. Adapting the concepts of continuous integration and continuous deployment become easy and best practice in microservices. Teams can deploy their microservices independently. If changes are made to a particular microsevice, that microservice can be tested and deployed in a stand alone fashion.

Having smaller codebases have a lot of benefits, it makes it easier to understand, modify and test the codex. Microservices enable organizations to minimize the number of developers working on a particular service, because each of these services can be run, maintained and managed independently.

One last benefit of microservices is the ease of replaceability. When the system is composed of lightweight bounded services, replacing one of them with a newer, better implemented codebases or even putting one of them in retirement is relatively an easy task. Microservices enable teams to be more comfortable with their codebases and provide them with more space for management and maintainability.

1.6 Microservices Disadvantages

All of the benefits discussed in the previous section don't come without a price. Microservices are not a silver bullet, they have their disadvantages as well. Microservices have all the inherited complexities of distributed systems. Things

that were simple to manage like transactions become distributed. Handling different technology stacks, platforms require more effort at the system level. With the distributed codebases, datastores and services, testing becomes much more complicated and the need for more modern tools become crucial. Another aspect is monitoring, monitoring a distributed system and identifying problems become more challenging compared to the old monolithic way. Logs, tracing and metrics become more complicated and harder to manage.

Another aspect that has become harder to manage is security. In a typical microservices architecture where a huge number of services communicate with each other, security becomes a more challenging factor as the focus will change from securing a single monolithic application or a set of finite few applications in an SOA, to securing a large number of applications that heavily communicate with each other to provide functionality [17]. Another security complexity is network security; microservices communicate with each other over the network, which adds the burden of network complexities and security to the plate [54].

1.7 The Lack of Research about Microservices Aspects

In the past years, both academia and practitioners helped explore, enhance and research microservices systems and their aspects, driven by the highly industrial adoption of it as an architectural style in enterprise and startup companies.

A lot of the underlying principles of microservices were thoroughly explored by researchers, but the research on the microservices themselves and their challenges is still far behind its rapid development and the challenges of the industry [62]. One of these challenges that attracted our attention is microservices security.

In microservices, security expands across multiple layers. Starting from the hardware layer, the cloud, communication, service, application as well as virtualization and orchestration [62].

In their systematic mapping study, Alshuqayran at el. stated that microservices systems security is an important challenge that is not well researched and needs more attention [3]. Among these security aspects are authentication and authorization. Our research focuses on this security aspect of authentication and authorization in the microservices architecture.

1.8 Available schemes for Authentication and Authorization

In microservice, there are multiple approaches to achieve authentication/authorization; centralized and decentralized:

- Centralized authentication - centralized authorization: Where a central point in the microservices architecture is responsible for authenticating and authorizing the API calls to the system.
  A simple example of centralized authentication - centralized authorization in a microservices architecture is to apply both authentication and authorization at the API gateway level only, any request that passes the API gateway will be authenticated and authorized before making its way to the destination microservice.

- Centralized authentication - decentralized authorization: Where authentication is the responsibility of a central point in the system while authorization is delegated to the requested microservice itself.
  An example of centralized authentication - decentralized authorization is

to apply authentication at the main entrance point of the microservices architecture; the API gateway while leaving the authorization checks at the microservices edges.

- Decentralized authentication - decentralized authorization: Where both authentication and authorization are left to the targeted microservice to handle in a decentralized fashion.

  An example of decentralized authentication - decentralized authorization in a microservices architecture is to apply both authentication and authorization at the microservice level, with no checks applied at a central point like the API gateway. In this case, the trust boundaries of the system are totally pushed inward to the microservice perimeter.

Each of these approaches are used to build microservices applications, the decision of which of these approaches is the best to pick depends on the nature of the application and the complexity of its provided functionalities. In the next chapters, we explain that the last approach is the most secured one in terms of security principles adoption and its fulfilment of the fundamental set of security requirements for a microservices architecture.

1.9 Research Objectives and Problem Statement

In this research, we are aiming to answer the following questions:

1. What are the security factors that would be used to apply the authentication and fine-grained authorization for systems that are deployed based on a microservices architecture?

2. What are the characteristics of a security framework that best suits security requirements of a microservices architecture in terms of authentication and fine-grained authorization?

3. What are the performance implications of the security framework in terms of latency overhead within the context of a microservices architecture that is based on HTTP RESTful API's?

1.10 Summary of contributions

The main contributions of the research:

- Determining the best architectural style for applying authentication and fine-grained authorization in a microservices system taking into consideration a proposed set of microservices security requirements.

- Proposing a new security framework for authentication and fine-grained authorization in a microservice based system.

- Proposing and conducting an experiment to measure the performance implications for the proposed security framework which will contribute for generalizing our approach so it can be applied for similar microservices cases.

In this thesis, we performed the following activities:

- Conducted a literature review and studied the related work in the field.

- Identified the application authentication and fine-grained authorization security goals for microservices architecture and identified the set of security requirements associated with these goals.

- Proposed a new security framework that fulfills these security requirements. Conducted a threat model analysis using the STRIDE threat framework to identify the potential threats and their risk reduction strategies. We used these strategies to enrich our security framework requirements.
- In order to prove the effectiveness of our framework, we applied an industrial motivating use case to our security framework (the Applicant Tracking System).
- Applied the proposed security framework to the industrial use case.
- Proved that our framework achieves the designed set of security requirements.
- Conducted an experiment and studied the performance implications for the proposed security framework.

1.11 Structure of this thesis

Chapter 2 discusses the main concepts around microservices security. It gives the reader a background about access control models, security tokens platforms, fine-grained authorization frameworks and security techniques we will use in our research. Chapter 3 discusses literature and state-of-the-art in microservices authentication and fine-grained authorization. We identify the main security goals in microservices application security. Finally, we highlight the limitations in the current practices. In chapter 4, we discuss the microservices security framework for authentication and fine-grained authorization and the industrial motivating use case. Chapter 5 outlines the research methodology that will be followed to generate, collect and analyze data. It also discusses the experimental

design. Chapter 6 describes the security framework implementation technologies and provides a demonstration of the security framework effectiveness. It also includes experiment evaluation and statistical analysis. In chapter 7, we discuss the key findings of our research and compare our work with the current state-of-the-art. In the final chapter, we provide a conclusion, future work and threats to validity.

Chapter 2

Background

In this chapter we discuss some concepts around microservices security that we will use throughout the study. We start by defining the well known access control models. After that, We give a brief description about the security token platforms. Because our research focuses on fine-grained authorization, we will explain to the user two fine-grained authorization frameworks that are used and well known in the industry, the "eXtensible Access Control Markup Language (XACML)" and the "Open Policy Agent (OPA|)". We finish the chapter by giving a background about two important security techniques we will use in our security framework, the sidecar for endpoint security and the usage of multiple trust domains.

2.1 Access Control Models

Many access control models have been developed across the past decades. The basic idea behind them is to restrict and organize access to the system information. In this section, we will discuss the major access control models: The Mandatory Access Control (MAC), The Discretionary Access Control (DAC),

The Role-Based Access Control (RBAC) and The Attribute-Based Access Control(ABAC) [26, 30]. All of these access control models are meant to provide authorization. Authorization can be divided into two main categories; coarse-grained and fine-grained. An example of a coarse-grained authorization is to allow a user from a certain group or who have a certain role to perform an action. An example of a fine-grained authorization is to allow a user to perform a specific action on a specific object constrained by a specific condition.

2.1.1 Mandatory Access Control (MAC)

MAC is mainly concerned with data confidentiality where access is given based on security labels. Policies are used in MAC to make a decision based on a previous set of configurations. MAC is usually controlled by a security policy administrator. Each user can be given explicit access to specific resources. In MAC, it is difficult to change accesses to reflect the changes in data and user accesses. MAC has been historically used in government and military applications.

A simple example of MAC will be in a government agency where security labels will be attached to all objects along with its classification information and each user will be given a clearance to what she can access.

2.1.2 Discretionary Access Control (DAC)

DAC is a traditional access control that allows users to control the access to their data. It allows access based on user identity, so it manages who can access what. It also deals with permission inheritance and auditing. By using DAC, the system can provide flexibility by maintaining a database of user identities and their permissions.

Relying on users defined accesses holds its risk. If there is a simple configuration mistake, wrong people may have access to certain data that they shouldn't have access to.

An example of DAC is adding, deleting or modifying permissions of a file on a Windows machine by its owner. At the discretion of the file owner, she can assign various permissions for different types of users.

2.1.3 Role-Based Access Control (RBAC)

In RBAC, users are assigned roles. Based on these roles, the user access is defined. The role can be expressed as a set of permissions that relate to a subject. Users can be assigned multiple roles at the same time. If an intruder got access on behalf of a user, the intruder access will be bound to that user access, she will not have access to the whole system.

A simple example of RBAC will be in an organization that consists of multiple departments. The system administrator can define a group for each department, if a new hire is on-boarded in the Human Resources (HR) department for example, the system administrator can add the new user to the HR group where she will inherit all the access and permissions assigned to this group.

2.1.4 Attribute based Access Control (ABAC)

ABAC added the dimension of request attributes. ABAC works with authentication, authorization and accountability. When the user requests an access to a resource, the access is controlled based on the request attributes, the user access and the destination resource. ABAC provides better security and flexibility compared to its predecessors. ABAC provides a hierarchical structure for permissions as well.

An example of ABAC is to limit the access of a specific website to a subset of users who have the HR role. This can be achieved by defining the website as an attribute, and only assigning this value to the specific users who will have access to it.

In this study, we focus on ABAC since it provides the concept of policies that can express complex rules and present more advanced scenarios and use cases compared to its predecessors: RBAC, DAC and MAC.

2.2 Security Tokens Platforms

In a microservice system, different application entities establish trust between each other by exchanging security tokens. Security token is usually issued by a trusted third party. When exchanging these tokens, cryptography is considered a basic building block both in the communication channels, by using Transport Layer Security (TLS) and within the generated token itself [24].

In this section, we discuss a set of token platforms: API keys, Open Authorization (OAuth2), OpenID Connect and JSON Web Tokens (JWT).

2.2.1 API Keys

An API keys is a simple encrypted token. It is used to call an API that does not involve user private data. API keys are usually used for clients who do not have a backend side such as web browsers and mobile applications. They are usually embedded within the application itself. The API key is used to track the requests generated from the client as a base of identification. Figure 2.1 shows the basic API Key usage scenario.

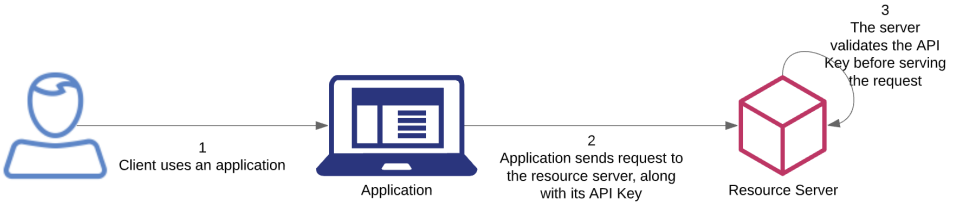

FIGURE 2.1: API Key Usage Overview

An important limitation of the API Keys is that they do not identify the user or the application who is making the request. The validation step only involves checking the validity of the issued API Key. If an attacker acquired the API Key, she can simply use it on behalf of the legitimate application. This is the main reason why API Keys are not recommended for authorization purposes.

2.2.2 Open Authorization (OAuth2)

Open Authorization (OAuth and the latest version of it OAuth2) is an industry standard protocol for authorization [43]. It focuses on the development simplicity in its flows. OAuth2 can be used in web applications, mobile applications and desktop applications as well. OAuth2 and its specification is developed by IETF OAuth Working Group [23] and defined in the RFC 6749 [21]. OAuth2 allows third party applications to have a limited scoped access to a resource on the owners' behalf. This is usually referred to as "access delegation".

OAuth2 defines four main roles:

- **Resource Owner**: the one who owns the resource.
- **Resource Server**: the server which hosts the protected data.
- **Client**: the application requesting access to the resource server.

- **Authorization Server**: the server which is responsible for issuing access tokens to the clients. These tokens are used to access the resource server.

Figure 2.2 shows the basic OAuth2 flow.

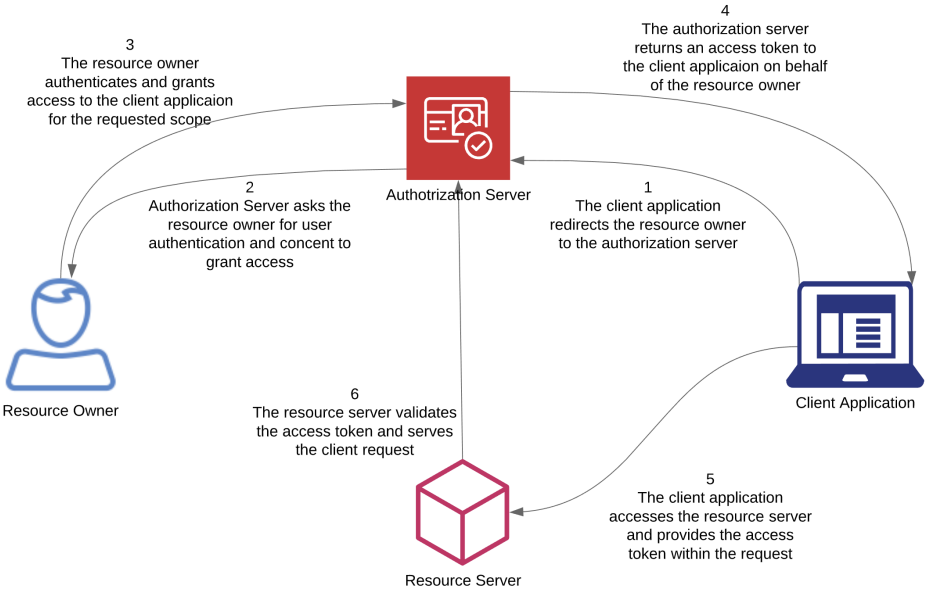

FIGURE 2.2: Overview of OAuth2 Flow

There are multiple delegation grant types in OAuth2:

- **The client credentials:** usually used for authentication between systems without the intervention of an end user.
- **The resource owner password:** This type of grant is suitable for trusted applications. The resource owner credentials are sent to the client application then to the authorization server to acquire an access token.
- **Authorization code:** this grant type is the appropriate one for end-user involvement, it allows to acquire a long lived access token on behalf of the resource owner.

- **Implicit grant:** similar to authorization code but it does not involve the step of getting an authorization code before acquiring an access token. This is why implicit grant type is not recommended. It allows access token leakage and access token replay.
- **Refresh token:** used to renew the expired access token.

2.2.3 OpenID Connect

OpenID Connect is an identity layer that can be added on the top of OAuth2 to provide identity of the end user. OpenID Connect uses the concept of ID token, which is a JSON web tokens (JWT) that contains the authenticated user information.

The usage of OAuth2 and OpenID Connect usually involve acquiring a single token from an authorization server. This token can be used to access different services and components on behalf of the user.

2.2.4 JSON Web Tokens (JWT)

JSON Web Token (JWT) is an internet standard defined in the RFC7519 [27]. It can be defined as a container that holds certain assertions from one place to another. The assertion is a valid strong statement issued by a trusted entity. JWTs are signed by the issuer private key and base64 encoded. Any receiver of a JWT can check its validity and decide whether to accept it or not.

The JSON Web Tokens has three main parts:

- The JOSE (JSON Object Signing and Encryption) header, which contains the metadata of the token, such as the algorithm used to sign the message.

- The payload, which carries a set of claims, such as the subject, audience, the time of expiration and the time when the token was issued.
- The signature, which is used to validate the token.

Figure 2.3 shows an JWT example for an encoded token and its decoded counterpart.

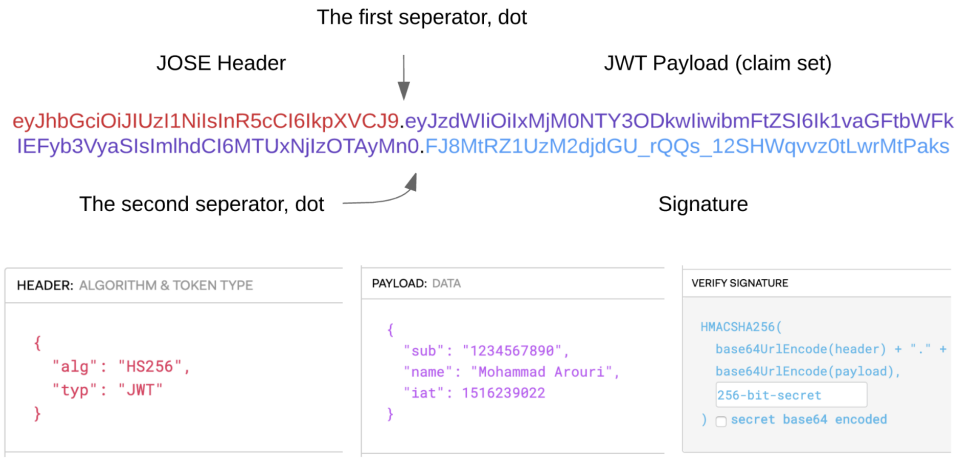

FIGURE 2.3: JWT Example

2.3 Fine-Grained Authorization Frameworks

In this section, we provide the main differences between a fine-grained and a coarse-grained authotization. Then we give a brief description of two well-known fine-grained authorization framework. The first is the XACML standard and the second is a promising modern framework, Open Policy Agent (OPA).

2.3.1 Comparison between Fine-grained authorization and Coarse-grained authorization

Granularity is defined as the quality of including a lot of small details [36]. It also refers to the extent a system is composed of distinguishable entities.

In his book "Guide to Computer Network Security", Kizza defined authorization granularity as the level of details that is required from an authorization entity to limit and separate privileges [29]. Coarse grained systems consist of few, large components, while fine grained systems consist of much more components of smaller sizes.

Authorization granularity has two main categories, coarse grained and fine grained. A clear distinguish between these two main categories is that coarse grained authorization policies can be very simple; as simple as “all or nothing”. While fine grained authorization policies provides the ability to distinguish among different entities and resources and can be much more complex compared to their coarse grained counterpart [22].

Coarse grained granularity only offers a basic ability to interact with system resources, all lower details within these resources functions are ignored. On the other hand, fine grained granularity provides very specific individual interactions to the defined tasks within the system resources.

We will illustrate this difference by an example, allowing only users of a certain group to perform a privileged functionality is an example of a coarse grained access authorization. While allowing an individual user to perform a specific action on a specified object is an example of a fine grained access authorization.

A real manifestation of a granularity example is as follows; suppose we have a system (applicant tracking system) where job seekers apply to opening jobs to be hired. Allowing a group of users (recruiters) to screen job seekers who applied to their opening jobs is a coarse grained access authorization example. Only allowing recruiters to screen job seekers within their department and restricting this screening activity within their working hours is an example of a fine grained access authorization.

2.3.2 The eXtensible Access Control Markup Language (XACML)

XACML is a security standard offered by OASIS back in 2003 [48]. XACML uses XML for defining policies and enforcing them. XACML is used for fine-grained, attribute-based access control. XACML standard contains multiple points, each of them has a specific functionality in the flow. The main points are Policy Decision Point (PDP), Policy Enforcement Point (PEP), Policy Administration Point (PAP) and Policy Information Point (PIP). Figure 2.4 shows the main data flow in XACML.

The Policy Decision Point (PDP) is responsible for evaluating the incoming requests, it compares the XACML request with its corresponding policy and returns a response for the caller. The Policy Enforcement Point (PEP) acts like a guardian to the resources. It asks the PDP for the authorization decision in order to allow or deny the request. Policy Administration Point (PAP) is where policies can be written and managed. It is also responsible for delivering these policies to the PDP. Policy Information Point (PIP) is the system entity responsible for providing attribute values for other points.

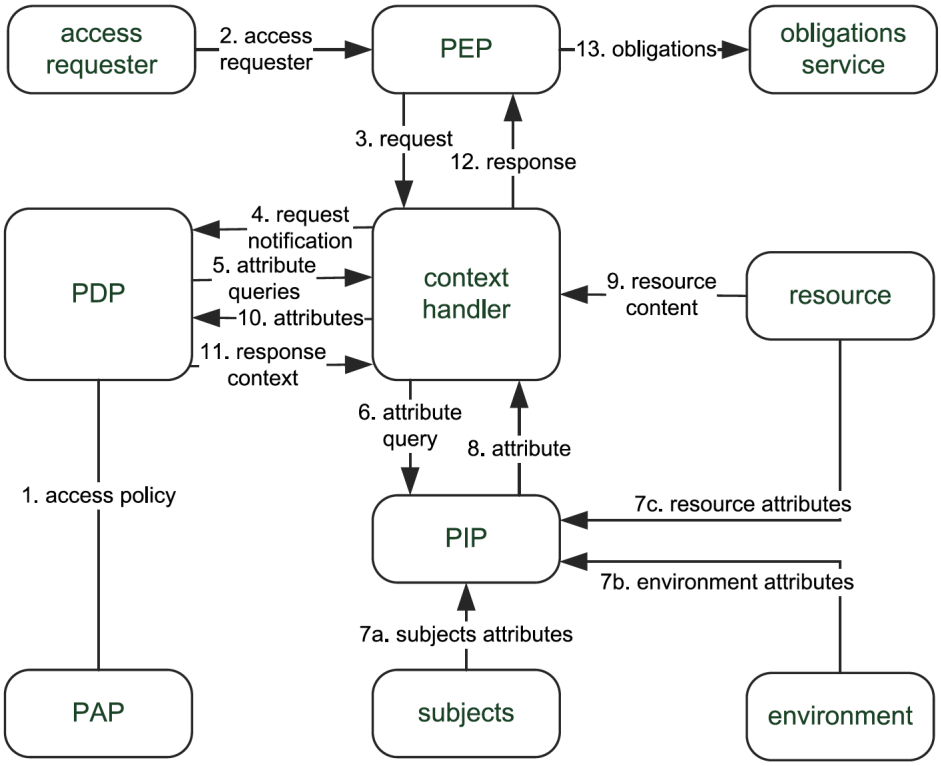

FIGURE 2.4: XACML Data Flow Model

2.3.3 Open Policy Agent (OPA)

A new emerging authorization framework is the Open Policy Agent (OPA) [44]. It is a Cloud Native Computing Foundation (CNCF) incubating project. OPA defines itself as an open source, general purpose policy engine mainly focused on policy enforcement.

OPA provides a high level declarative language (Rego) and simple APIs to offload the policy decision point from the application services. Its design is a good fit for cloud services and microservices.

OPA capabilities are similar to XACML, in which it has a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP) as well as Policy Administration Point (PAP). One of the core principles of OPA is to decouple the PDP from the PEP. The software can query OPA and supply the appropriate data structures as input to the engine. The engine will generate a decision by leveraging the query, policies and data. Figure 2.5 depicts this flow.

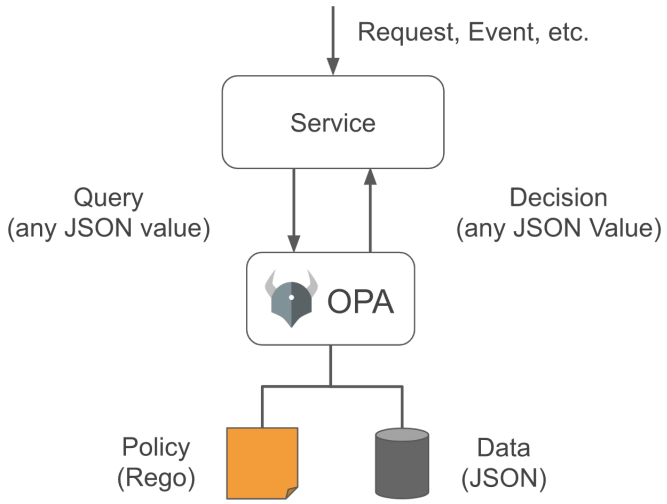

FIGURE 2.5: Open Policy Agent (OPA) Data Flow Model

2.4 Emerging Security Techniques

In this section, we describe two security emerging techniques we that use and address in our study, the sidecar for endpoint security pattern and the usage of multiple trust domains.

2.4.1 Sidecar for Endpoint Security

With the increasing adoption of cloud solutions, the need for security principles increases, among these adopted principles are the “zero trust network” and the

"least privilege" principles. The zero trust network basically implies that never trust the network, either internal or external and always verify. The "least privilege" states that systems should not give any resource more permissions than needed. Each job should be assigned the minimal set of permissions, no more. These two principles will be a good guidance for our security decisions along the way.

Sidecars for security is an emerging security technique that enforces these principles at every microservice endpoint. The sidecar is a process or a container running alongside the resource microservice. It can intercept the incoming traffic to the resource microservice and act like a policy enforcement point (PEP). Open Policy Agent (OPA) and Envoy are popular frameworks that can be deployed as a security sidecar.

2.4.2 Multiple Trust Domains

Multiple trust domains can be found in many microservices systems. These trust domains can be defined on the basis of the teams they are managing them, or on the basis of governance issues or even organizational boundaries. The larger and more complex the microservices system, the chances to find multiple trust domains are more likely.

From a security perspective, multiple trust domains mean multiple authorization servers. The microservices in the same trust domain trust security tokens generated by its authorization server. When a microservice wants to communicate with another microservice in a different trust domain, it needs to acquire a security token that is trusted by the second domain before initiating the call.

One of the big advantages of using different trust domains is to minimize the effect of token theft. Each security token can only access one trust domain. The security token will be considered invalid if it used outside its original trust domain. This basically means, if a token is stolen, the attacker can gain access only to the resources belongs to the trust domain who generated the access token and not to the entire system. Tokens that have access to the entire system without any limits are usually called "powerful tokens".

2.5 Connecting the Dots

On one hand, having such a wide set of access models, security tokens platforms and standards provide a wide set of choices for developers and architects to build systems with high quality. On the other hand, some of these tools is more appropriate than others.

Microservices tend to offer fine-grained APIs for greater reusability across the system [62], these APIs need a corresponding fine-grained authorization framework that protects the resources and data provided by these APIs. This explains the push toward ABAC access model more than RBAC or the traditional old models of MAC and DAC.

When talking about security tokens, most companies seek the use of the defacto standard OAuth2 rather than using a proprietary security framework. OAuth2 has a lot of support and a wide set of implementations across different programming languages and frameworks. Following a well known standard eases the integration between microservices system and other systems as well as with external clients.

OAuth2 works well for authentication along with static and coarse-grained authorization purposes. The fast advances of business needs and the push for more security controls created the need for a more fine-grained authorization framework. This is where XACML and OPA play a great role. These frameworks can be integrated in a microservices system on top of OAuth2 to provide a fine-grained access control. Despite that XACML is a well known, relatively old and mature standard, it does not see a wide adoption in the industry. On the other hand, OPA seems a more promising authorization platform that is built from ground up to help modern applications secure their applications in an easy and efficient way.

In our research, we focus on the ABAC access control model. We use OAuth2 and JWT as a way of exchanging and verifying security tokens with OPA for more advanced fine-grained authorization.

2.6 Summary

In this chapter, we discussed the main concepts of microservices security, starting from access control models, security token platforms to fine-grained authorization frameworks. In the next chapter, we conduct a thorough literature review and study the microservices and their security aspects in literature. During this critical review, we identify a set of security goals for microservices security. We follow that by a set of existing security frameworks that handle authentication and fine-grained authorization, their strengths and their weaknesses.

Chapter 3

Literature Review

In this chapter, we conduct a thorough literature review where we discuss microservices and their security aspects found in literature. We explore a set of high level, strategic security goals that researchers focused on to address microservices security challenges. We finish the chapter by discussing three existing security frameworks that target authentication and fine-grained authorization security in microservices, their strengths and weaknesses. We compare each of these frameworks with the security goals identified previously.

3.1 Microservices in Literature

Since the early show of the microservices, researchers kept an eye on its industrial advances and added their touch to its state-of-the-art. Many researchers focused on the techniques, best practices and lessons learned from transforming monolithic applications into a microservices based architecture [12, 35, 18, 58]. Other aspects that brought the attention of the researchers includes resource management, service composition, data management, portability and security [15].

Since we are focusing on security aspects of microservices, the next section discusses the microservices security in literature. What are the security dimensions and layers the researchers focused on.

3.2 Microservices Security in Literature

In their research, Dragoni at. el. discussed the challenges microservices introduced due to their open nature [16]. The promotion of services reuse enabled developers and application builders to integrate with third party services added a set of new challenges. One of these challenges is to provide proper authentication and authorization for the microservices system.

Nehme at. el. also discussed the openness nature of microservices and the trust challenges caused by this [39]. They discussed microservices security from multiple dimensions: microservices components, application architecture, securing infrastructures in terms of operating systems, network and securing external interfaces for interdomain communication. They stated that every microservice should be treated as an independent component which has its own security measures and can not be trusted by other parties.

In literature, microservices security has many categories. Each of them has its own choices and decisions. Yarygina al. el. proposed a hierarchical decomposition for these security layers [62].

The first layer is the hardware layer, it takes care of the security modules at the hardware level. The second layer is the virtualization layer, where security aspects of isolation and sharing of libraries and hardware caches are addressed.

The next layer is the cloud layer, which discusses the security issues of the hypervisors and the various remote attacks. The fourth layer is the network layer, it addresses the standard communication protocols, such as TLS as well as security integration styles. The fifth layer is the Service/Application layer, which discusses security issues of error handling, input validation, protection for data at rest and programming languages security vulnerabilities. The last layer is the orchestration layer, where security issues of services discovery and registry are addressed.

Security is a crucial aspect in software development, insecure applications put critical infrastructures at risk. With the latest advances of web application development and the advantages developers and companies are gaining from cloud solutions and services, more complex and connected software solutions are being introduced. These advances made application security a difficult task to achieve and maintain. Modern application security requires awareness, protection against common security risks and the ability to discover and resolve these risks in a quick and efficient manner.

3.3 Microservices Security Goals

Microservices architecture has a lot of security challenges. Some of these challenges are certainly not new, they were introduced and discussed in SOA. While some of these challenges still apply to an exact extent in microservices, other security challenges become more complex given the characteristics of the microservices. In this section, we will discuss the security goals researchers focused on to address different microservices security challenges. These goals are high level, strategic goals which our system should achieve [57]. In chapter 5, we return to these goals and derive from them both requirements and assumptions.

3.3.1 Defend the Greater Attack Surface

In a monolithic application, a single host contains the entire application. The attack surface in such a system is limited and contained within the perimeter of its operating system and the security concerns will be focused on this singular point. While in microservices, where the application is split into many services that communicate to each other using APIs, independently from the microservices internals and even programming languages, a more broader attack surface is introduced. In addition to that, the openness nature of microservices and their need to communicate with each other as well as with the outside world. This became a game changer for microservices where they increase the need for more advanced security solutions that are more suitable to the newly introduced challenges [16].

In a microservices system, the attack surface can be divided into two main parts, the system boundary or the outside perimeter and each microservice boundary. The system boundary defence were discussed by Jander at. el. [25]. They stated that the system tries to protect itself from the unwanted outsider actions. The microservices architecture enlarges this perimeter and made it harder to monitor, manage and secure. This attention to the great perimeter sometimes implies that security of the individual microservices is underestimated or even neglected.

In their research Yarygina at el discussed the redefinition of the security perimeter [62]. They explained that microservices redefined this perimeter and pushed it inward toward the boundaries of each microservice.

In order to achieve our goal of defending the greater attack service of microservices, we explored the literature for the best security principles to adapt. Among these principles is the "Defence in Depth" principle.

Defence in depth implies the concept of adding multiple security mechanisms on the different levels of the system. In order to do this, security checks are applied at each microservice level instead of having them at a central place of the system, like the API Gateway.

3.3.2 Handle Network and Communication Complexity

Decomposing a monolithic application into microservices can easily result in the creation of hundreds of microservices. A good example of this is Netflix [59], where the decomposition of their monolithic application resulted in the creation of hundreds of microservices.

These microservices still need to communicate with each other, exchange information and share states. This introduced a lot of complexities on the network layer and added complexities to handle network failures, debugging and auditing. The increase of such complexities opened a new opportunities for attackers to perform attacks against the system [2].

Traditional communication attacks like eavesdropping and man in the middle attack (MITMA) are examples of common attacks in microservices. In order to build a secure authentication/fine-authorization framework. We need to make sure the underlying network and communication layer is secure. In order to do so, we will use the standard protocols of Transport Layer Security (TLS) and Mutual Transport Layer Security (mTLS) will be used.

Mutual authentication or two-way authentication is a solution that aims to establish trust between the two parties. In our case, between two microservices. By default, TLS only one way, where the client can prove the identity of the server. TLS also offers client to server authentication using clients certificate [56]. Client certificates requires certificates provisioning and involve less user friendly experience, so it is usually avoided to be used in applications. In mTLS, each of the parties can prove the authenticity of the other side. mTLS prevents replay and man in the middle attacks [55].

There are multiple solutions that eases the use and management of mTLS in a microservices architecture, docker swarm [34] and the Secure Production Identity Framework for Everyone (SPIFFE) [52] are good examples.

3.3.3 Establish Trust between Microservices

In the early stages of microservices, software engineers tend to neglect the security concerns of the inter-communication between the different microservices. Microservices are usually designed to trust each other in an open way. This practice imposes big security risks, such as confused deputy attack, where the attacker gaines illegitimate access to one of the microservices, then uses its privileges to access the resources of other microservices claiming the identity of the compromised microservice. In a system with open trust, this may result in a compromise at the overall all system level.

The attacker can hold multiple identities. On one hand, she may be an external actor who is trying to gain illegitimate access to the system. On the other hand, she may be an insider who is abusing her privileges to gain sensitive information or control other microservices [54].

One of the principles we will adapt is the "Zero Trust" principle. Zero trust implies that the security paradigms should move from the system boundaries inside toward the individual microservices and resources. A zero trust architecture implies that no implicit trust should be granted to any resource in the system [50].

In order to establish trust between microservices, we will use security tokens in our framework. When using security tokens, microservices can establish trust with another internal microservice or with an external actor by exchanging security tokens. These tokens are usually generated by a third trusted party. Security tokens can be classified into two main categories; traditional and modern web tokens.

Web sessions are a form of traditional security tokens. Web sessions have been used for a long time, mainly as a proxy for user authentication. When a user identifies herself to the system, by providing her username and password, the system grants a session token that the browser can save and send in subsequent requests as a means of user authentication.

Web sessions come with a security cost, they have known limitations that can exploited by attackers such as session hijacking and cross-site request forgery [14]. Because sessions act like a powerful token, If an attacker acquires a valid user session, she will be able to do all the actions granted to that user within the system.

The new modern security tokens are more API centric, simple to adapt, implement and mostly JSON based. Modern frameworks overcome old security

issues by establishing a new approach to handle security. Some of these practices are the avoidance of using the password as a powerful primary authentication factor, decoupling application functionality from security authentication and authorization and the use of short lived security tokens.

These modern tokens are widely used as a means of authentication and authorization for both applications and users. API keys, OAuth2, OpenID Connect and JWT are the main modern platforms used for this purpose [37].

In the next section, we study the existing security frameworks suggested and implemented by researchers in the field of microservices security to solve for authentication and fine-grained authorization. This is followed by identifying the gaps and limitations of these current practices.

3.4 Authentication/Fine-Grained Authorization Security Frameworks

In this section we discuss three proposed security frameworks, which targeted authentication and fine-grained authorization security in microservices. We compare each of them with the security goals identified previously. We also discuss the limitations of these frameworks.

Yarygina at. el. discussed the unique landscape of microservices and the emerging security practices in the field [62]. They suggested a security framework that suits a microservices based architecture. The framework relied on the use of a certificate authority for mTLS and a reverse security token service to issue security tokens per user request. The authors tested their proposed framework against a toy microservices-based system (MicroBank).

To defend greater attack surface of their system, the authors adapted the defence in depth by redefining the perimeter security.

They also used mTLS and principal propagation to achieve trust between microservices and handle network security complexities. They suggested the use of a new JWT per request. Generating a security token per request can be overwhelming from a performance perspective. The authors stated that their security solution degraded the overall system performance by 11%, 4% for the mTLS usage and 7% for the use of security tokens. In their suggested model, the usage of the security tokens were barely for authentication purposes. The solution lacks the real existence of a fine grained authorization.

The second framework we want to mention was suggested by Davy Preuveneers and Wouter Joosen. They suggested a security framework to handle the issue of delegated fine-grained authorization policy in a microservice based system [46]. The authors investigated a microservices based data processing workflow (a healthcare system) from a dynamic authorization point of view. How microservices can collaboratively contribute to answer an authorization data decision achieving a good level of application security and forbidding illegitimate access from gaining access to the system.

In order to achieve this, the authors depended on two microservices basic elements: feature toggles and circuit breakers. Feature toggles enables the gradual integration of new features into the system. While circuit breakers aim to protect system workflows from unauthorized access.

The authors did not discuss the issue of greater surface attack. But the appliance of security authorizations check at every microservice endpoint implicitly implies that their model is a defence in depth enabled model.

Network and communication where out of the focus of their study as well. The authors only discussed the security authorization policies and their ability to be evaluated in a distributed manner. They adapted the least privilege principle. When a microservice wants to make an authorization decision that needs the involvement of other microservices. It asks these services to evaluate their part of the policy, without the need to access or pull the original data from these participant microservices. This enabled a distributed policy evaluation with minimal data exposure.

In order to achieve a fine-grained authorization system, the authors used their own simplified version of XACML [46]. The authors stated that using XACML is complex, so they introduced a lightweight XACML policy language for authorization policy definition. The lightweight policy language was based on JSON instead of XML, it leveraged only a subset of the feature set in XACML with both simplifications and limitations. They dropped policy targets and policy composition to more simplify their prototype framework.

The use of XACML introduces complexities to the system in order of defining and managing policies. The authors attempt to introduce a new abstracted lightweight version can not be generalized to be used on a wide scale due to its prototyping nature, lack of support and limitations.

The third security framework was proposed by Nehme at el. that targets key security challenges of microservices [38]. They focused on providing a fine-grained authorization framework by using both OAuth2 and XACML. The authors didn't discuss the perimeter defence explicitly. But the usage of a local API Gateway at the front of each resource microservice implies the adoption of defence in depth principle. TLS were used to mitigate against network and communication complexities. So their solution lacks the capability of the authenticity check of the caller.

To establish trust between microservices, Nehme at el. used security tokens in terms of OAuth tokens between the different internal and external calls. The authors used an OAuth client for every pair of microservices. This basically means the flow will return to the user for notification and consensus in each new microservices integration. This can be overwhelming especially if the interacted microservices are within the same trust boundaries. Our proposed solution will take the multiple trust domains into consideration. It will use the same security token for microservices within the same trust boundary. This will minimize the resource owner intervention for similar microservices and simplify system flows without decreasing the overall system security.

3.5 Summary

In this chapter, we discussed microservices security in literature. We explored a set of high level, strategic security goals that researchers focused on to address microservices security challenges, and finished by discussing a set of existing security platforms for microservices authentication and fine-grained authorization. In the next chapter, we propose a new security framework that handles

authentication and fine-grained authorization in microservices and discuss its details.

Chapter 4

Microservices Security Framework for Authentication and Fine Grained Authorization (MSFAA)

In this chapter, we propose our security framework and discuss its details. In order to achieve this, we start by confirming that security is the main quality attribute we are taking into consideration represented by the security requirements. We also focus on the performance of the proposed framework. The performance of any proposed security solution should be taken as a crucial deciding factor. Bad performing solutions are not practical and usually abandoned by developers and practitioners. Taking the performance into consideration gives our model more credibility. We evaluate the performance in terms of API latency overhead against a proposed industrial motivating use-case; the Applicant Tracking System (ATS).

Next, We define the functional requirements of the ATS, model the microservices and design the REST APIs. This system represents our null-architecture.

After that, we design our security model and discuss its components. We explain how our proposed security checks are applied to meet the security requirements. Next, we add sequential diagrams to address the inner details of the framework objects interactions.

4.1 Motivating Use Case, The Applicant Tracking System (ATS)

Job seekers spend a lot of time navigating between different job boards, searching for an appropriate job to apply for. Finding the correct job is like finding a needle in a haystack. After finding a feasible job, the job seeker spends minutes manually filling information before submitting the application. This process repeats each time the job seeker wants to apply for a new job, no matter if this is the first time apply on this system, or its an old system that she used before.

On the other side of the system resides the businesses, businesses use Applicant Tracking System to receive job applications for their opening jobs. They seek talent and best match job seekers. This involves manually navigating through submitted resumes, categorizing and filtering applications, which take time and effort and can be considered error prone.

This application is a type of "Software as a Service" (SaaS) and is a typical fit for microservices architecture. Using a microservice architecture to implement this type of application provides it with the necessary quality, scale, availability and maintainability.

4.1.1 Functional Requirements

Table 4.1 shows the functional requirements for the Applicant Tracking System (ATS).

TABLE 4.1: Applicant Tracking System (ATS) Functional Requirements

ID	Actor	Functional Requirements
R1	Job Seeker	Create account
R1	Job Seeker	Enrich account using resume parse
R1	Employer	Create account
R1	Employer	Create payment subscription
R1	Employer	Publish a job
R1	Job Seeker	Apply for a job

Figure 4.1 shows the basic use cases for our motivating application.

4.1.2 Modeling Microservices

After the functional requirements were specified and the use cases were identified. The next step is to model our microservices. In this step, we grouped the related functional requirements to form the bounded contexts. These bounded contexts will define the microservices.

The system will contain three main microservices:

- **The Account microservice:** it will be responsible for job seeker registration, enriching job seeker profile by providing the resume parse functionality and the business registration.
- **The Marketplace microservice:** it will be responsible for the job lifecycle. Publishing a new job by an employer will push it into the marketplace. Job seekers can search and apply for existing jobs.

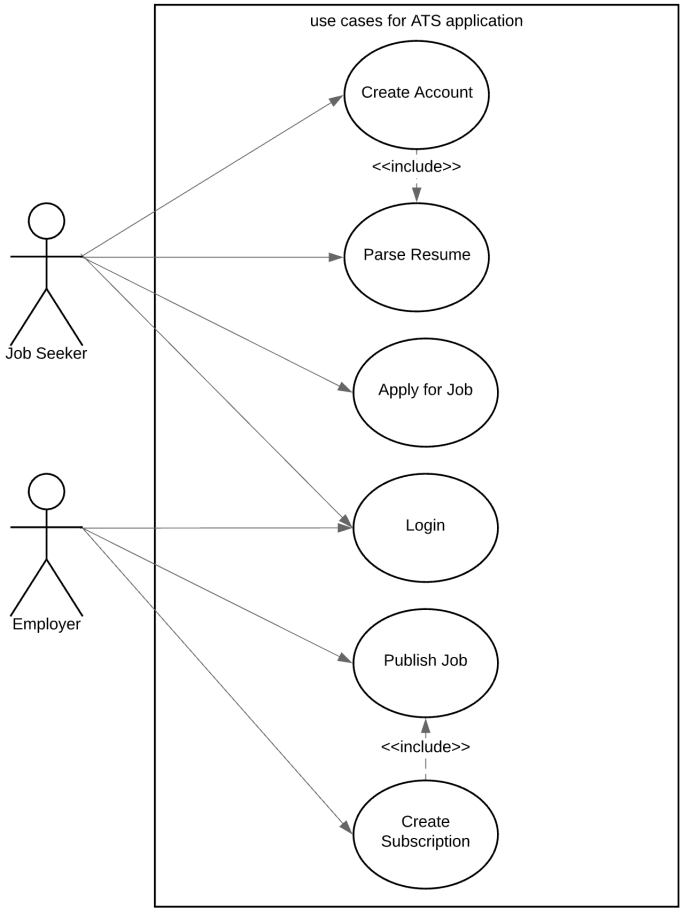

FIGURE 4.1: Applicant Tracking System (ATS) Use cases

- **The Financial microservice:** it will responsible for business subscriptions. Employers can create a new subscription.

Figure 4.2 shows the basic architecture for the Applicant Tracking System (ATS) (the null-architecture). Detailed description about the used symbols can be found in Appendix A.

The architecture contains the following basic elements:

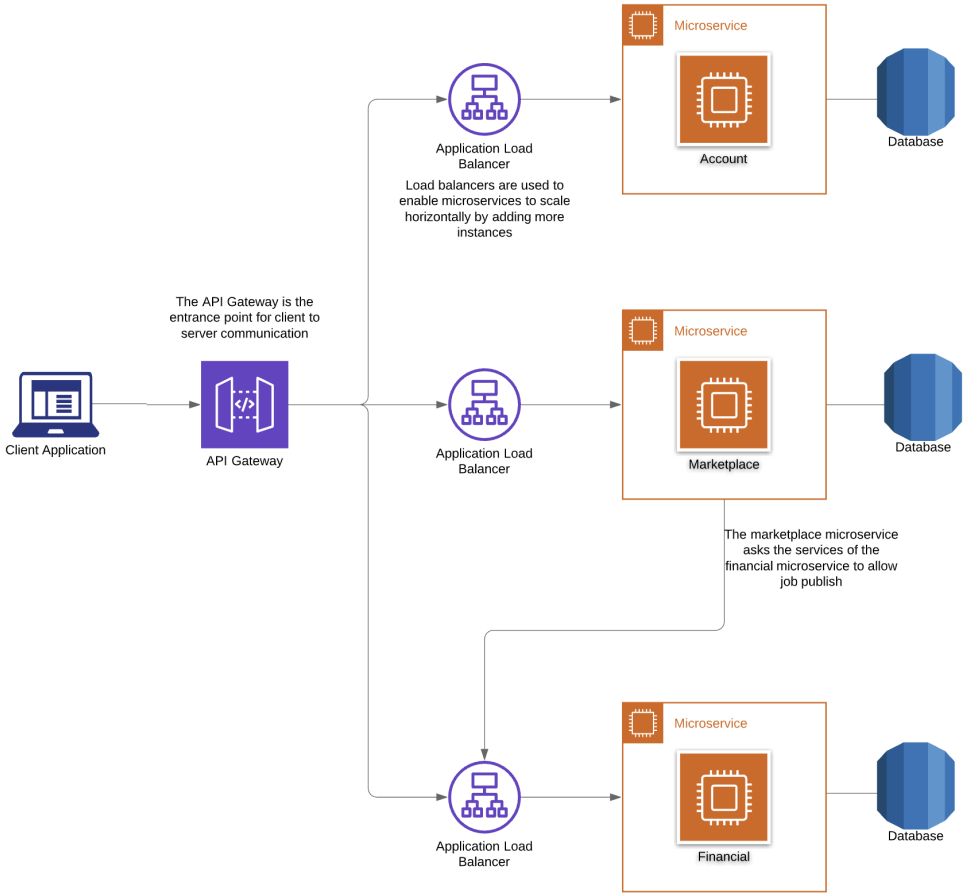

FIGURE 4.2: Applicant Tracking System Architecture (the null-architecture)

- **The client application**: The ATS services are offered using a client application. This application can be a web client or a mobile client (iOS, Android, ..).

- **The API Gateway**: Represents the entry point for external actors to the system.

- **Load Balancers**: load balancers are used to distribute the incoming traffic into the existing microservice instances. Load balancers enable horizontal scalability for the application. This gives the microservice the advantage

of avoiding the single point of failure and enables it to increase throughput in an efficient manner.

- **The microservices**: The application consists of three microservices as described earlier; account, marketplace and financial microservices.
- **The databases**: each microservice has its own datastore, where it persist its data in its own bounded context.

4.1.3 REST APIs Design

Table 4.2 shows the HTTP REST APIs that will be exposed by each of the microservices in our application.

4.2 Security Requirements and Assumptions

In this section, we will describe our security requirements and assumptions. Those items under the responsibility of the system in our software-to-be will form our requirements. While those items under the responsibility of the system in the environment will form our assumptions [57].

4.2.1 Security Requirements

Following are the main security requirements, we use these requirements in the design phase, each of them contributes to the security measures and checks that we add to the proposed framework.

- SecurityRequirement-1: The system should have the capability of giving its administrators controlled access by using policies. These policies are fine grained (fine-grained access/defence in depth).

TABLE 4.2: Applicant Tracking System (ATS) REST API Design

HTTP Method	API Signature	Requirements	Microservice
POST	/users	Register a new user	Account
POST	/businesses	Register a new business	Account
GET	/users/id	Get user profile details	Account
POST	/users/id/resume/parse	Parse a resume for a job seeker to enrich profile	Account
GET	/jobs	Get available jobs to apply for	Marketplace
POST	/users/id/jobs/id/apply	Job seeker applies for a job	Marketplace
POST	/businesses/id/jobs	Create a new job for a business	Marketplace
POST	/businesses/id/jobs/id/publish	Publish an existing job	Marketplace
GET	/businesses/id/jobs	Get available jobs for a business	Marketplace
GET	/businesses/id/subscriptions	Get active subscriptions for a business	Financial
POST	/businesses/id/subscriptions	Create a new subscription for a business	Financial

- SecurityRequirement-2: Accessing personal identifiable information needs data owner approval. No personal data can be accessed without user approval and consent (fine-grained access/defence in depth).
- SecurityRequirement-3: Data exchange between services should be limited and follows a predefined rules for service consumption (zero trust/defence in depth/).
- SecurityRequirement-4: Security as a non-functional requirement should be treated as an aspect to the service itself. In other words, the security concern should be handled separately from the application logic and the typical functional requirements held by it.
- SecurityRequirement-5: Avoid the use of powerful tokens (zero trust/defence in depth).
- SecurityRequirement-6: Mitigate against token theft attack (zero trust/defence in depth).

4.2.2 Security Assumptions

Traditional security mechanisms that are out of the scope of authentication and fine-grained authorization are considered out of the scope of our study. This includes intrusion detection systems, intrusion prevention systems, encryptions and security for data at rest.

4.3 The Security Model

In this section we describe our security model main service and describe each service role in the overall model and how it contributes to the original security

requirements. We also show the security standards that we described in the previous chapters which are used in our model.

4.3.1 Security Model Services

Our proposed framework consists of a group of fundamental security services that need to be up and running. These services are:

- Certificate Authority Service.
- Authorization Server.
- API Gateway.
- Sidecar.
- Security Trust Domains.

We discuss each of these components main functionalities and how each of them contributes to the security requirements.

4.3.1.1 The Certificate Authority Service (CA)

The CA is the first part of the suggested security model. It enables the mTLS between the different microservices. The goal of this entity is to ensure a secure communication between the different microservices and to enable the different microservices to prove the authenticity of the other. The CA enables the system to prevent both replay and man in the middle security attacks.

4.3.1.2 The Authorization Server

The Authorization Server is the second service in our model. Its main responsibility is to generate OAuth2 security tokens in the form of JWT. These tokens are

temporal (short lived) and have a contained narrow access that is limited to one security trust domain. In our use case, we will need two security token services, one for each of the domains.

4.3.1.3 The API Gateway

Common models of microservices use API Gateways. From a security perspective, gateways have multiple advantages like security scanning, throttling, Intrusion Detection System (IDS), Web Application Firewall (WAF).

In our model, the API Gateway acts as a main central entry point for external parties, minimizing the microservice architecture attack surface. The API gateway will transform the opaque token received with the incoming request with the corresponding JWT token. Any request received without a valid opaque token will be rejected at the API Gateway level.

This way, the API Gateway will be able to globally validate each external API call. If the system invalidated an access token, this token will be evicted from the tokens cache and the API gateway will reject all requests holding that token. The simple goal of the opaque token here is to hide the details of the JWT from the external parties and to get a centralized place to manage external parties calls to our system.

4.3.1.4 The Sidecar for endpoint Security

The sidecar is the fourth component in our security model, Security roles of the sidecar are:

- Decouple the security as a non-functional requirement from other business logic functional requirements that are contained and runned by the microservice.
- Provide the ability to be configured, redeployed and managed without interfering with the original microservice functionality in an ad-hoc style.
- Acts as a Policy Enforcement Point (PEP). Sidecars can check the security tokens and apply a fine-grained authorization policies before passing the requests to the resource microservice.

Benefits of using sidecar design pattern:

- One of the security requirements was to deal with the security as an aspect in our system. Which basically means that security implementation should not be mixed with the applications logic that is maintained by the microservice itself. A reusable and configurable sidecar gives us the ability to deal with the service security with ease and simplicity. Configurations and policies can be added, changed and removed without affecting the microservice itself.
- The sidecar should behave as a single entry point to the microservice. This will give us the ability to apply our security measures on a central point with the guarantee that no access will be performed on the microservice without the intervention of the sidecar.
- Applying a sidecar for security maximizes portability. With better portability the value of the design increases.

Figure 4.3 shows a typical microservice with a sidecar running in front of it. All incoming calls are intercepted by the sidecar first, then passed to the destination resource.

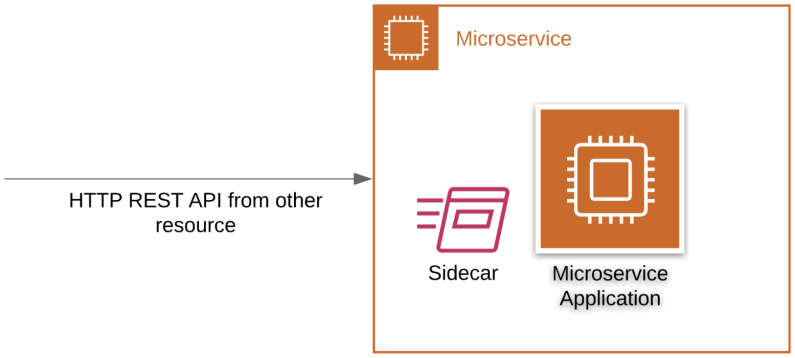

FIGURE 4.3: Microservice with a Sidecar Overview

4.3.1.5 Security Trust Domains

In section 2.4.2, we explained the need for defining multiple trust boundaries. The use of security trust boundary is a fundamental piece in our model. It minimizes the effect of token theft and mitigate against the use of powerful tokens. In our toy microservice system, there are two security trust domains. The first one contains two microservice; the Account and the Marketplace services. The other one contains the Financial microservice.

4.3.2 Security Standards

Our suggested security framework depends on existing security standards, following are the main security standards/frameworks we will use:

- Open Authorization (OAuth2).

- JSON Web Tokens (JWT).
- Open Policy Agent (OPA).

4.3.3 Security Model Architectural Diagrams

Figure 4.4 shows our the basic flow of a client accessing a microservice, the client needs to acquire a valid OAuth2 token from the authorization server. Then it can access the resource microservice using the acquired security token. The resource microservice sidecar checks the validity of the token before allowing access to the resource.

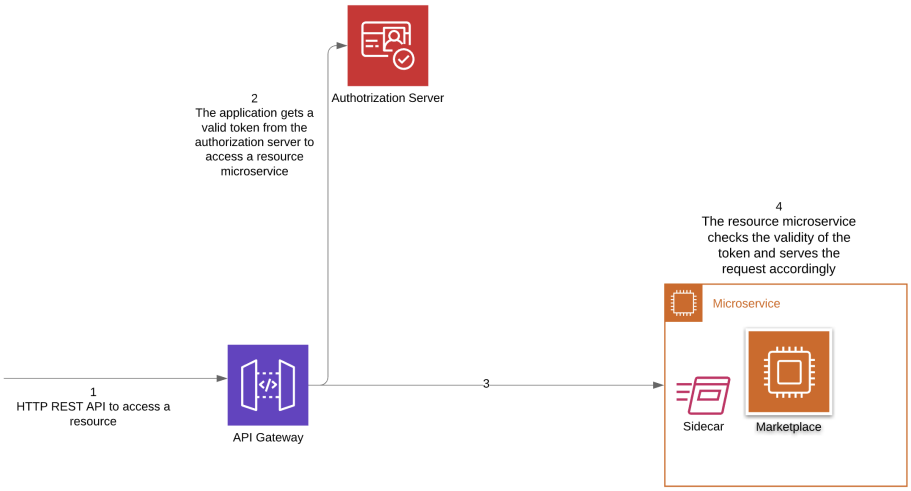

FIGURE 4.4: Client Accessing a Microservice Flow

Figure 4.5 shows a simplified view of our proposed security architecture for microservices within the same domain.

Figure 4.6 shows an architectural diagram that contains all the security components, the API Gateways, Authorization Servers, Sidecars and the two security trust domains.

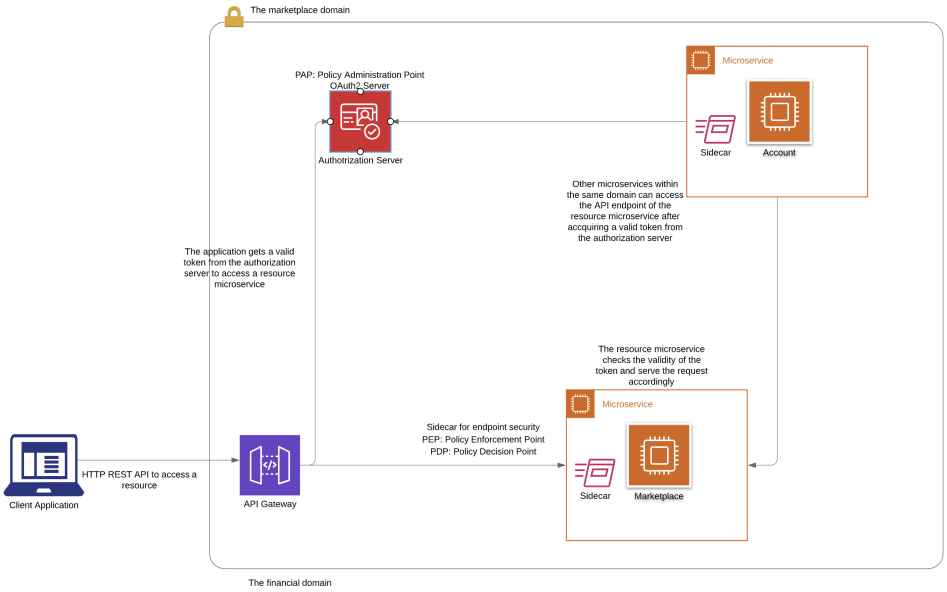

FIGURE 4.5: Security Framework Architecture for One Trust Domain

4.3.4 Security Checks and the Appliance of Security Requirements

After explaining the main services and providing the architectural diagrams of the proposed security model. We discuss how these services achieve the security requirements and what are the proposed security checks to achieve that. We start by explaining the flow within the same trusted domain for both internal and external API calls. Then we explain calls between different trusted domains.

For internal API calls (from a microservice to another microservice):

- For a microservice to call another one, it first needs to acquire an access token from the authorization server. The authorization server will check for the validity of the provided credentials as well as for the scope of the requested token [SecurityRequirement-1][SecurityRequirement-3]

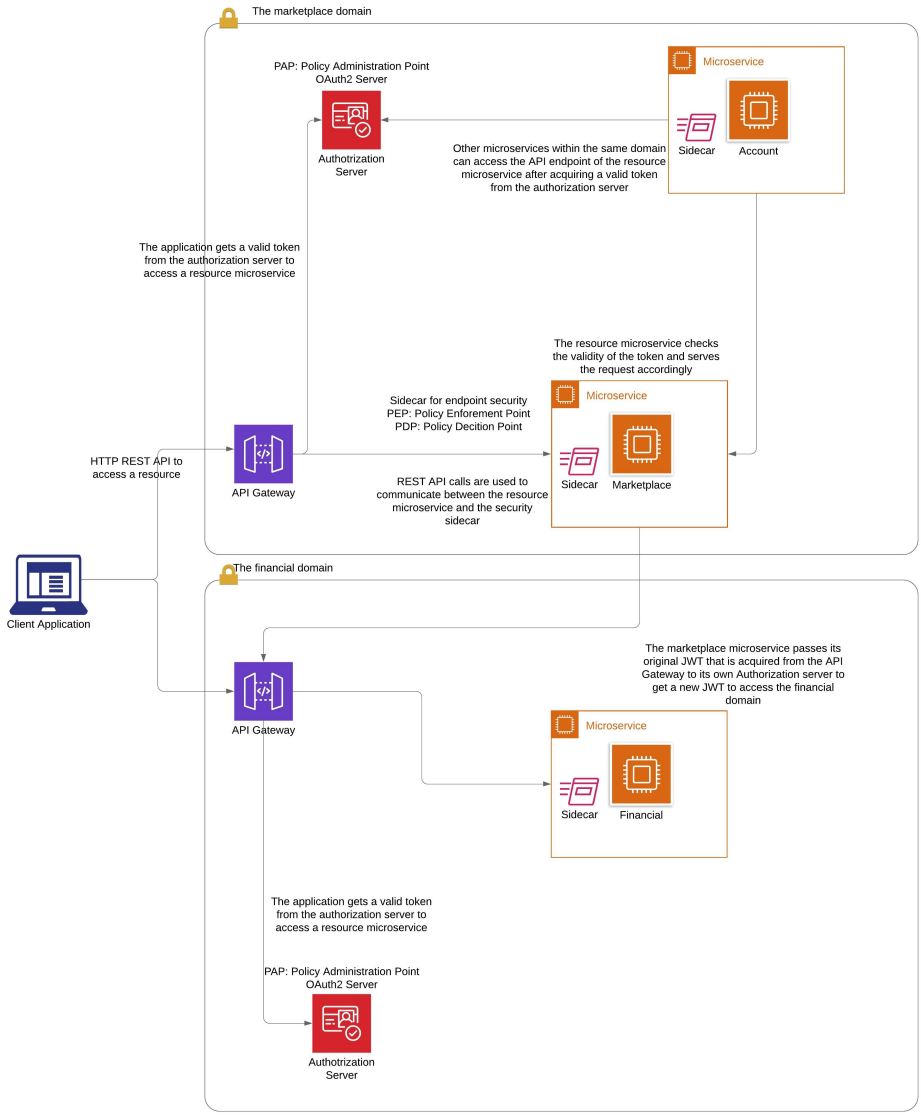

FIGURE 4.6: Security Framework Architecture for Multiple Trust Domains

- After that, the caller microservice will use the acquired security token to initiate a call to the resource microservice. After the request being received

by the microservice, the security sidecar will be the first responder to the request [SecurityRequirement-4]. It will check for the authorization access key. If it does not exist, it will return an unauthorized response to the caller. If the authorization token exists. The sidecar will check the passed JWT. The JWT will contain the scopes, subject identifier and the expiry date [SecurityRequirement-2]. At this step, the JWT check is a basic check only, the main purpose is to check that the subject is authenticated. The fine-grained authorization part will be performed in the next step.

- The next check is the fine-grained authorization check, where the sidecar will query OPA and provide the user, action, method and path. OPA will return the decision, this decision will be interpreted by the sidecar. If the decision is permit, the request will be passed to the resource microservice. Otherwise, it will be rejected and the token will be invalidated [SecurityRequirement-1].

For external API calls (from a client to a resource microservice):

- The first check will be performed by the API Gateway, where the check for authorization header will be performed. If the header does not exist, the API Gateway will return an unauthorized response to the caller. If the authorization token exists, the API Gateway will check the validity of the opaque token and replace it with a JWT.
- After this, the request will be forwarded from the API Gateway to the microservice, the same set of checks from the previous section (For internal API calls) will be applied to the request.

Next, we explain the security checks between two different trusted domains.

For internal API calls (from a microservice to another microservice):

- Security tokens that are issued by an authorization server are only valid within the trusted domain of the issuer authorization server. If a microservice within a trusted domain wants to call another microservice in another trusted domain, it needs to exchange its JWT with a new JWT from the resource microservice authorization server that is trusted by the second trusted domain API Gateway and resource microservice. Having an access token trusted only by one trusted domain is a mitigation against the use of powerful tokens[SecurityRequirement-5]. No single access key can access the entire system. If a token was stolen by an attacker or by an intruder, the token will only have access to the resources of the issuer authorization server for a short period of time, which is typically set to one hour, the token will expire after this time window passes [SecurityRequirement-6].

- Upon receiving this request, the resource (destination) microservice will check the passed JWT that contains the scopes, subject identifier and the expiry date. If it does not match its rules, the request will be rejected [SecurityRequirement-3].

For external API calls (from a client to a resource microservice):

- The external client needs to acquire an access token per trusted domain. Each token is expected to have different scopes and privileges. If this token is stolen, it will be only valid for one trusted domain and for a limited period of time (until the token expires or the theft is detected and the token is invalidated) [SecurityRequirement-6].

4.3.5 Security Model Sequential Flows

In this section, we show the main sequential flows in our security framework. Figure 4.7 shows the flow to acquire a new authorization token (access token).

Figure 4.8 shows the various security checks for a regular request originated from a client application (i.e. web browser). Figure 4.9 shows the events when a microservice in a domain needs to access a resource in different trusted domain on behalf of a resource owner.

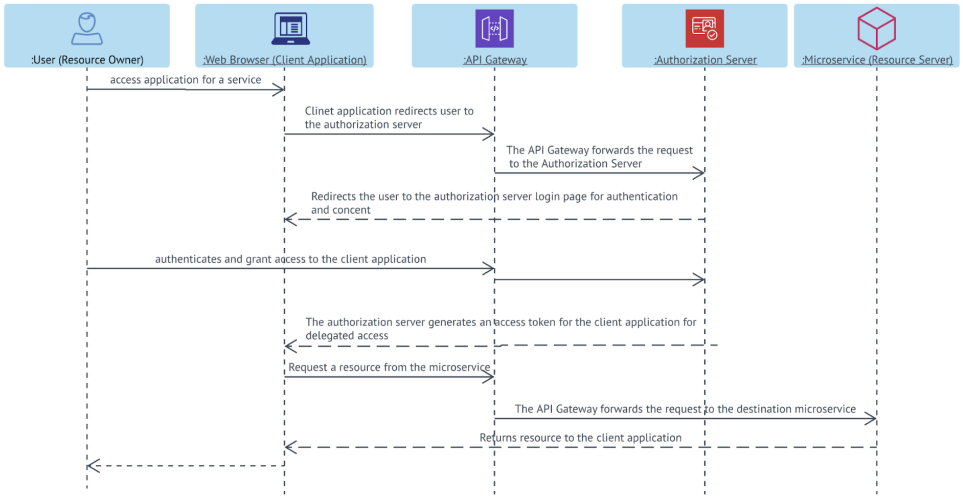

FIGURE 4.7: Acquiring Access Token Sequence Diagram

Figure 4.8 shows a sequence diagram for the security checks performed when a client application (i.e. a web browser) accesses a resource microservice.

Figure 4.9 shows the sequence of events for a microservice to access another microservice in a different trusted domain.

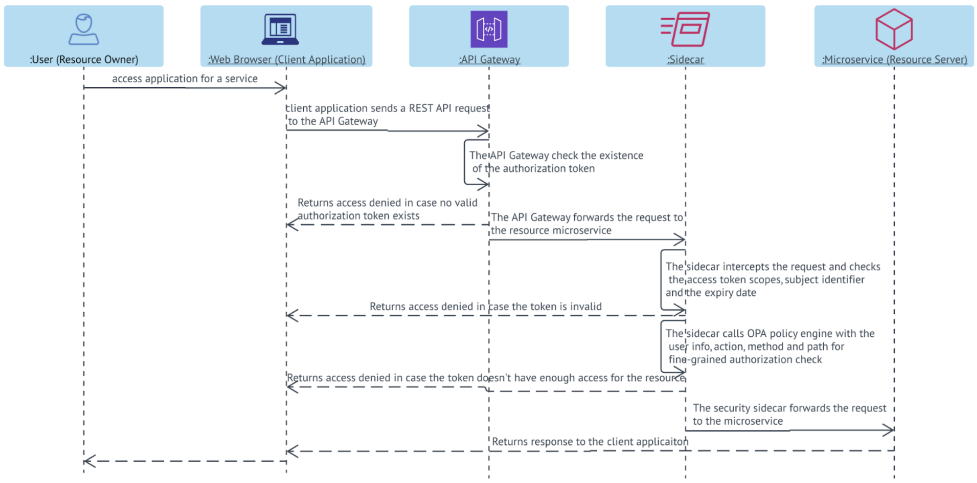

FIGURE 4.8: Security Checks for a Client Request Sequence Diagram

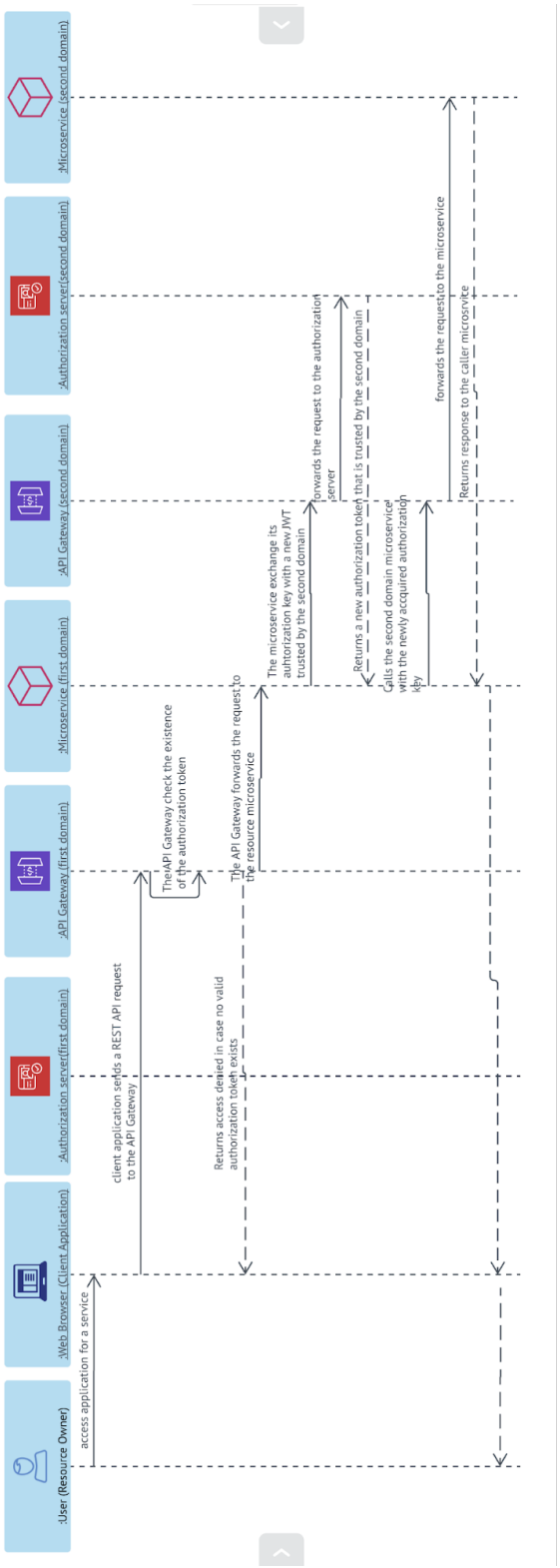

FIGURE 4.9: Security Checks for a Microservice accesses another Microservice in Different Trust Domain on behalf of a Client

4.4 Threat Model

In this section, we will explain our security threat model. Threat models help identify potential security threats and their risk reduction strategies. In order to generate our threat model, we followed a three phases process in which we started by analysing the system requirements and creating a data flow diagram (DFD). After that, we applied the STRIDE threat modeling framework to the data flow diagram to find the potential issues. In the third phase, we applied security controls to mitigate against the explored security issues.

The design phase was the starting point of our threat model. The security framework requirements were described in detail in Security Requirements section. Figure 4.10 shows the data flow diagram for our system. It shows a typical microservices system with the basic processes involved. It also includes the processes that represents the proposed security framework.

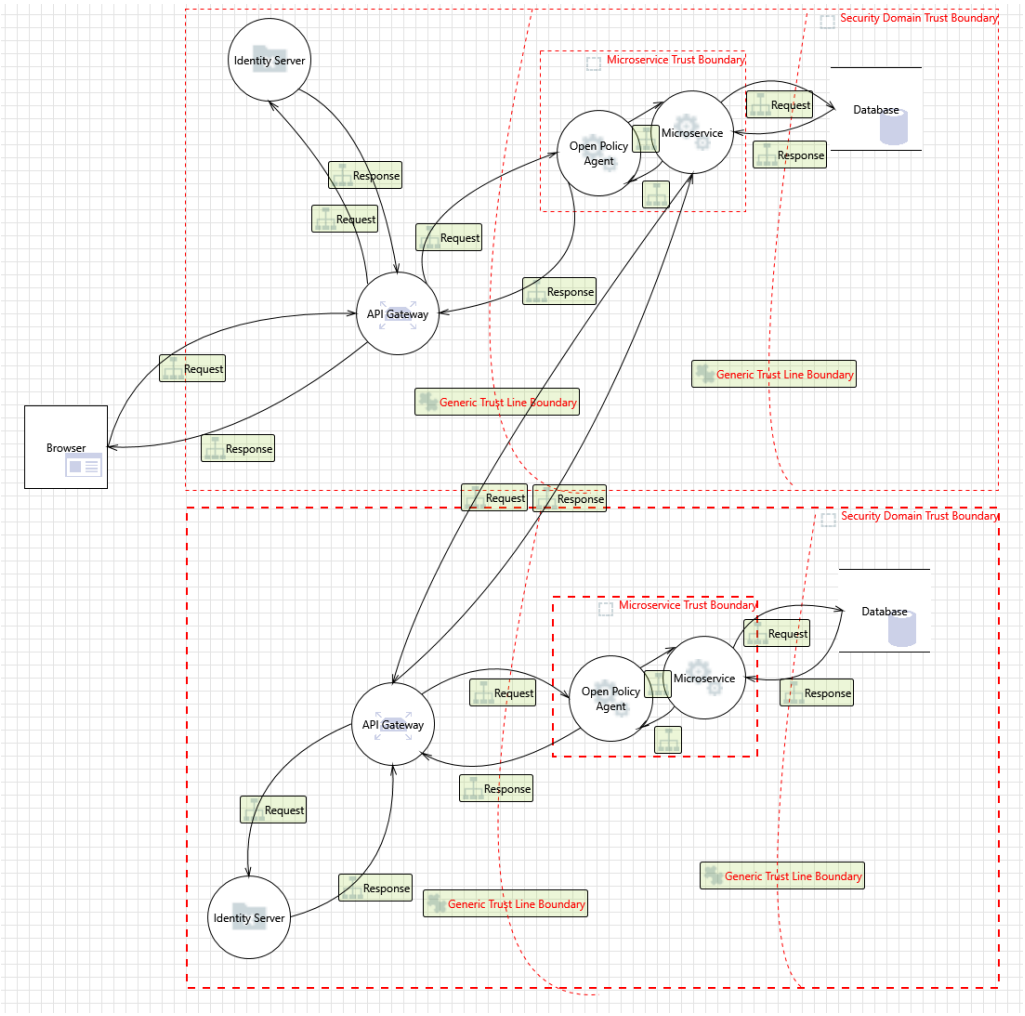

FIGURE 4.10: Threat Model Data Flow Diagram (DFD)

STRIDE is a threat modeling framework developed by Loren Kohnfelder and Praerit Garg at Microsoft, the main goal of it is to identify various threat types [33]. STRIDE examines these threats from the attacker perspective. IT is designed to be used to discover security vulnerabilities in a software system [51]. STRIDE classifies threats into six main categories:

- **Spoofing Identity**: Where an attacker pretends to be somebody or something else.

- **Tampering**: Where an attacker modifies data without in a malicious unauthorized way.
- **Repudiation**: Where an attacker denies performing an action in which other parties can neither confirm nor deny.
- **Information Disclosure**: Where an attacker is exposed to data which they are not supposed to have access to.
- **Denial of Service**: Where an attacker tries to bring the system down to deny or degrade the service to the legitimate users.
- **Elevation of Privilege**: Where an attacker gains an increased access which she shouldn't have.

We used STRIDE to identify potential threats then we applied a set of counter measurements to mitigate against each of these threats (the fix phase). Following are a description for each of these threats, its priority and counter measurements.

4.4.1 Threats and Counter measurements

In this section, we show the main set of security threats identified using STRIDE. For each of these threats, we provide a description, a priority level, a category that shows to which of the six STRIDE categories this threat belongs to and a set of counter measurements that is implemented in our security framework to mitigate against the identified threat. One point to mention is that we focused on the authentication and fine-grained authorization aspects of the application when mitigating against the identified set of threats, other mitigation techniques that are out of the scope of authentication and fine-grained authorization were not addressed in our study.

4.4.1.1 Network Threats

Sniffing or eavesdropping	An attacker can use a packet sniffer to read the traffic content of API calls between the user and the destination microservice. This may happen outside the network perimeters of the system if the calling user is an external user. It can also happen when a microservice calls the services of another microservice either in the same security trust domain or in another one.
Priority	High
Category	Information disclosure
Counter measurement	Encryption for data and security tokens in transit using TLS is a counter measurement to this attack.

Man in the middle attack	This attack happens when an attacker deceives the downstream microservice claiming that it is a legitimate host.
Priorit	High
Category	Spoofing
Counter measurements	- The usage of encrypted security token generation between the client and the authorization server mitigates this attack. - The usage of TLS encryption algorithm for communications between the different parties

4.4.1.2 Host Threats

Threat	Common host threats like footprinting, viruses, worms and arbitrary code execution may result in an attacker gaining an illegitimate access to a generated security token that she can use to access unauthorized data.
Priority	High
Category	Elevation of privileges
Counter measurements	Avoid the use of powerful tokens. Generated access tokens are temporal and short lived for only one hour.

4.4.1.3 Application Threats

Replay Attacks	The attacker captures the user's access token using a sniffing or monitoring tool. Then she uses it to gain access under the stolen identity.
Priority	High
Category	Spoofing
Counter measurements	Encryption for data and security tokens in transit using TLS is a counter measurement to this attack. Generated access tokens are temporary and short lived, although this countermeasure does not prevent the attack, it narrows the time window that the attacker can use to exploit the system.

Elevation of privilege	In this attack, the attacker tries to elevate her privileges to a more powerful account to gain more control over the system.
Priority	High
Category	Elevation of privileges
Counter measurement	The usage of a fine-grained authorization framework enables the appliance of the least privilege principle.

Disclosure of Confidential Data	This attack can happen if the application discloses confidential or sensitive data to an unauthorized user.
Priority	High
Category	Information disclosure
Counter measurement	The usage of a fine-grained authorization framework adds checks to every microservice call before allowing the operation to access the confidential data.

Data Tampering	This attack occurs when an unauthorized attacker modifies system data.
Priority	High
Category	Tampering
Counter measurement	The usage of a fine-grained authorization framework adds checks to every microservice call before allowing the operation to edit data.

4.4.1.4 Parameter Manipulation

Security token manipulation	security tokens are susceptible to modification. They are exposed on the client side and can be manipulated by an attacker.
Priority	High
Category	Tampering
Counter measurement	Each access token is verified and checked in the authentication process. JWT holds a signature that can be used to check if the token was tampered or not.

4.5 Summary

In this chapter, we proposed a new security framework for microservices authentication and fine-grained authorization. We discussed its security requirements, assumptions and flows. In the next chapter, we discuss our research methodology and experimental design. We also discuss how we measure the security framework performance.

Chapter 5

Methodology and Experimental Design

In this chapter, we explain our research methodology and the phases we followed during the research experiment. After that, we present the base design of our experiment. We start by explaining the research hypothesis, followed by a description of the independent, dependent as well as the neutralized variables. Finally, we show how we measured the performance of the proposed security framework in terms of API latency overhead.

5.1 Research Methodology

In order to check to what degree our proposed security framework serves the security requirements of a microservices architecture and to measure the performance implications of this framework, we conducted an experiment in which the workflow shown figure 5.1 will be applied. The workflow consists of four major phases:

- Setup microservices environment and deploy the applications.
- Apply workload and generate results.
- Extract and combine data.

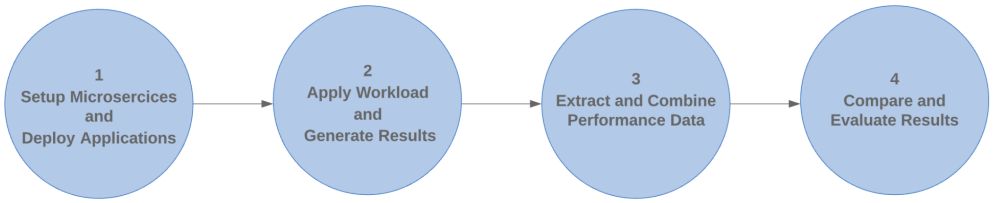

FIGURE 5.1: Experiment Workflow

- Compare and evaluate results.

In the first step, we need to define and design our proposed security framework. This has been discussed in detail in chapter 4. Where we defined our motivating use case; the Applicant Tracking System (ATS). We also defined our null-architecture as well as our security framework services, components, architectural diagrams, security requirements and checks. Next, we set up the microservices, the security framework and deployed the applications.

In the second step, we applied the workload on both the null-architecture at first. Then on the microservices architecture with the proposed security framework applied to generate results. One of the basic workload techniques that we followed to generate results is the linear workload, where we simulated an increase in the amount of users who are accessing the service to generate the results.

The main objective of applying the workload is to:

- Check the appliance and effectiveness of our proposed security checks against the security requirements.
- Gather performance data in terms of latency overhead.

We combined the results in the third step. In the last phase we followed an evaluation plan that consists of the following steps:

- Exploration: Different scenarios were presented and evaluated that shows typical and non typical scenarios.
- Validation: A group of scenarios that were identified in the previous step were validated. We focused on proving the these scenarios are repeatable, generalizable and have a similar behavior for similar simulations.
- Comparison: In this step we compared the set of results against each other to draw and extract results. The performance was measured in terms of latency overhead.

5.2 Experiment Design

This section presents the base design of our experiment. It starts with the research hypothesis, followed by a description of the independent, dependent as well as the neutralized variables. As mentioned before, the main goal of this experiment is to propose a security architecture for authentication and fine-grained authorization in a microservices system, based on a set of predefined security requirements. Then to study the impact of such proposal on the performance of the architecture in terms of API latency overhead.

5.2.1 Research Hypothesis

We have two main hypotheses. The main goal is to support the alternative hypothesis by rejecting the null hypothesis.

- **Null hypothesis:** There is no difference between the baseline architecture (the null-architecture) and the proposed architecture with security checks

applied on effectively securing the system in terms of authentication and fine-grained authorization.

The second null hypothesis: There is no difference between the baseline architecture (the null-architecture) and the proposed architecture with security checks applied on the performance of the system (measured by API latency overhead).

- **The alternative hypothesis:** There is a difference between the baseline architecture (the null-architecture) and the proposed architecture with security checks applied on effectively securing the system in terms of authentication and fine-grained authorization.

  The second alternative hypothesis will be: There is a difference between the baseline architecture (the null-architecture) and the proposed architecture with security checks applied on the performance of the system (measured by API latency overhead).

5.2.2 Dependent Variable

As we focus on security, securing the system in terms of authentication and fine grained authorization by the applied proposed security checks will be our dependent variable. We are also focusing on performance. The performance of the applied security model measured by the API latency overhead will also be a dependent variable in this experiment. The API latency overhead is a continuous variable and we will measure its values in milliseconds.

5.2.3 Independent Variables

In our experiment we have two independent variables, the security framework and security trust domains. The security framework is a binary categorical

variable with the values of either with-security framework applied or without-security framework applied. The second independent variable is the security trust domains, which is also a categorical variable that has a value of one global domain or more. In our experiment, we looked into a system that has one global domain or two security domains. The behavior of the systems that have more than two security domains is just like the system with two security domains in terms of security framework appliance and impact.

Combining the two independent variables will give us four treatment groups.

- Without the security framework on one security domain.
- With the security framework on one security domain.
- Without the security framework on two security domains.
- With the security framework on two security domains.

Each subject in our experiment will be a simulated user API that is originated by the Apache JMeter. With this setup, we have a completely randomized design since each subject is assigned randomly to a group. Another point to mention is that each subject will receive only one level of the experimental treatment. No simulated users will be shared between the different treatments.

5.2.4 Neutralized Variables

There are many variables that might affect the results of our experiment. To make our results valid, these variables have to be neutralized and controlled as much as possible. We can categorize these variables into four categories:

1. The infrastructure layer (the hardware layer).

    - Variable: Operating System
    - Value: Amazon Linux 2
    - Rationales: Amazon Linux 2 is based on Linux kernel 4.14 and tuned for optimal performance on Amazon EC2. It provides a secure, stable, and high performance execution environment to develop and run cloud and enterprise applications [8]. We used Amazon Linux 2 as the operating system of the microservices in our experiment.

2. The virtualization layer.

    - Variable: Server
    - Value: AWS EC2
    - Rationales: We used Amazon Web Services in our experiment. AWS provides reliable, scalable and inexpensive cloud computing services. It is also a leader in public cloud services [9]. Using AWS added more control and transparency on the server resources we used.

    - Variable: Server Resources - Memory
    - Value: 1 GB RAM, 2 GB RAM
    - Rationales: All of our microservices had 1 GB of RAM while the Apache JMeter was configured on an EC2 server with 2 GB of RAM to have more power and available resources than the downstream microservices.

- Variable: Server Resources - Disk space
- Value: 8 GB EBS(Amazon Elastic Block Store).
- Rationales: EBS is an easy to use block store service provided by AWS. It is natively supported with Amazon Elastic Compute Cloud (EC2 [7]).

- Variable: Server Resources - CPU
- Value: 2 AMD EPYC vCPU's
- Rationales: AMD EPYC is a high performance processor that has been built using Advanced Micro Devices company's Zen Architecture. AMD EPYC are cost effective and provides good performance [5]. They are suitable for workloads that do not need high sustainable compute power but experience temporary spikes in usage. Which is a good fit for our experiment.

3. The communication layer.

   - Variable: Network
   - Value: AWS VPC (Virtual Private Cloud) Internal Network
   - Rationales: To minimize the network effect, all of our servers were hosted inside one AWS VPC hosted in us-east-1 (N.Virginia) region. All the traffic between different components of the system were within this VPC.

4. The Application layer.

   - Variable: Application Programming Language
   - Value: Java

- Rationales: We used Java to build our industrial use case microservices. Java is well known programming language and has a good community base in building applications for microservices architecture.

- Variable: Application Framework
- Value: Spring Boot 2.2.4.RELEASE
- Rationales: following the use of Java programming language, we used Spring boot to build our microservices applications.

- Variable: API Gateway
- Value: AWS API Gateway [60]
- Rationales: One of the industrial solutions for the API Gateway is AWS API Gateway. It is easy to use and configure gateway that can provide us with the needed functionalities to conduct our experiment.

5.2.5 Measuring Performance

In our experiment, performance implications is an important aspect we are taking into consideration. We measured the performance of the proposed four treatments, before and after applying our security framework on one and two security trust domains. We measured the performance of these treatments in terms of latency overhead.

After setting up the environment and the microservices (the first phase described in the methodology). We applied a workload and generate results.

To apply a proper workload we used Apache Jmeter simulation tool to initiate a set of different users accessing the system. One of the most popular tools to generate loads is Apache JMeter. Apache JMeter is an open source tool, built using Java programming language and designed for load testing and for performance measurements [10].

We used Apache JMeter to generate a set of API requests representing a pool of users accessing the system. One important metric we measured is the latency overhead, which can be defined as the total time a single API request takes to go from the Jmeter as a starting point to its destination microservice and back again to the JMeter.

Figure 5.2 shows our basic Apache JMeter setup. We installed the JMeter on an AWS EC2 instance and used its command line tools to apply the workload. The results were collected for further analysis and evaluation in the later phases.

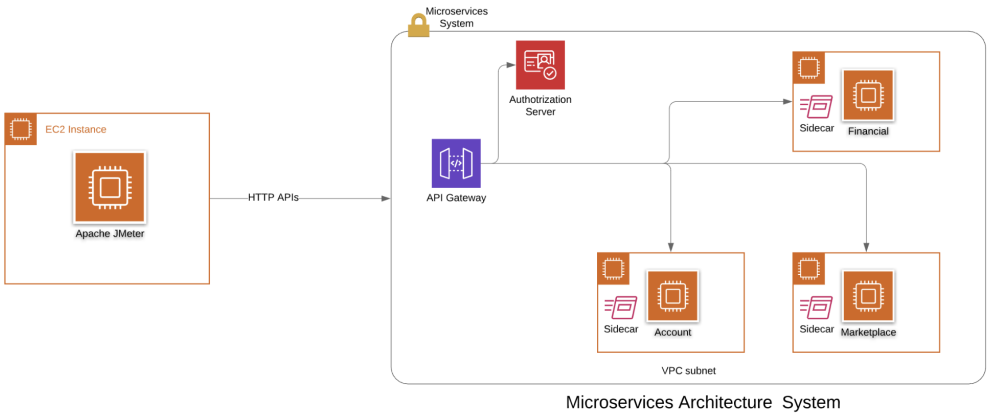

FIGURE 5.2: Apache JMeter Setup Overview

5.3 Summary

In this chapter, we discussed our research methodology and the phases we followed during the research experiment. We also presented the experimental design of the experiment. We finished the chapter by explaining our method of measuring the performance implications of the security framework. In the next chapter, we present the implementation technologies of the both the security framework and the null-architecture. Then, we put the security framework in action to verify its effectiveness. This be followed by an evaluation for the framework performance and verification to prove the statistical significance of our experiment.

Chapter 6

Experiment

In this chapter, we dive into the details of our experiment, starting by the implementation details of the null-architecture. After that, we describe the implementation details and technologies used to implement the security framework. Our security framework is not tied to a specific programming language or implementation framework, any programming language can be used as long as it fulfills the security requirements of the framework. Next, we provide a proof of the effectiveness of the security framework by putting it in action; we demonstrate how the security checks applied by the security framework fulfils the proposed security requirements and mitigate against security threats. After that, we discuss the experiment runs, initial results and how we checked the significance of the results.

6.1 Implementation Technologies

In this section, we describe the technologies we used to build our null-architecture and the technologies we used to build the security framework components. Any technology can be used to implement both the microservices and the security framework components, taking into consideration that the implementation

should respect the various security components of the security framework and their capabilities when choosing from the available technologies.

A feasible framework implementation should support OAuth 2, policies for fine-grained authorization, central API Gateway for external authentication, side-car for endpoint security and security trust domains.

Following are a list of the main components involved in the experiment and their implementation technology details.

6.1.1 Microservices

We used Spring boot 2.2.4 on top of Java 11.0.5 to develop our three microservices. Along with gradle as a dependency manager and tomcat as an application server. The codebase can be found under the following three Github repositories:

- Marketplace https://github.com/MArouri/Marketplace
- Financial https://github.com/MArouri/financial
- Account https://github.com/MArouri/Account

Figure 6.1 shows the project structure for the Marketplace microservice.

In order to ease the development operations, we built a CI/CD (continuous integration/continuous deployment) pipelines using AWS code pipelines, code deploy, auto scaling groups and target groups. All of this code can be found in the Github repositories as well.

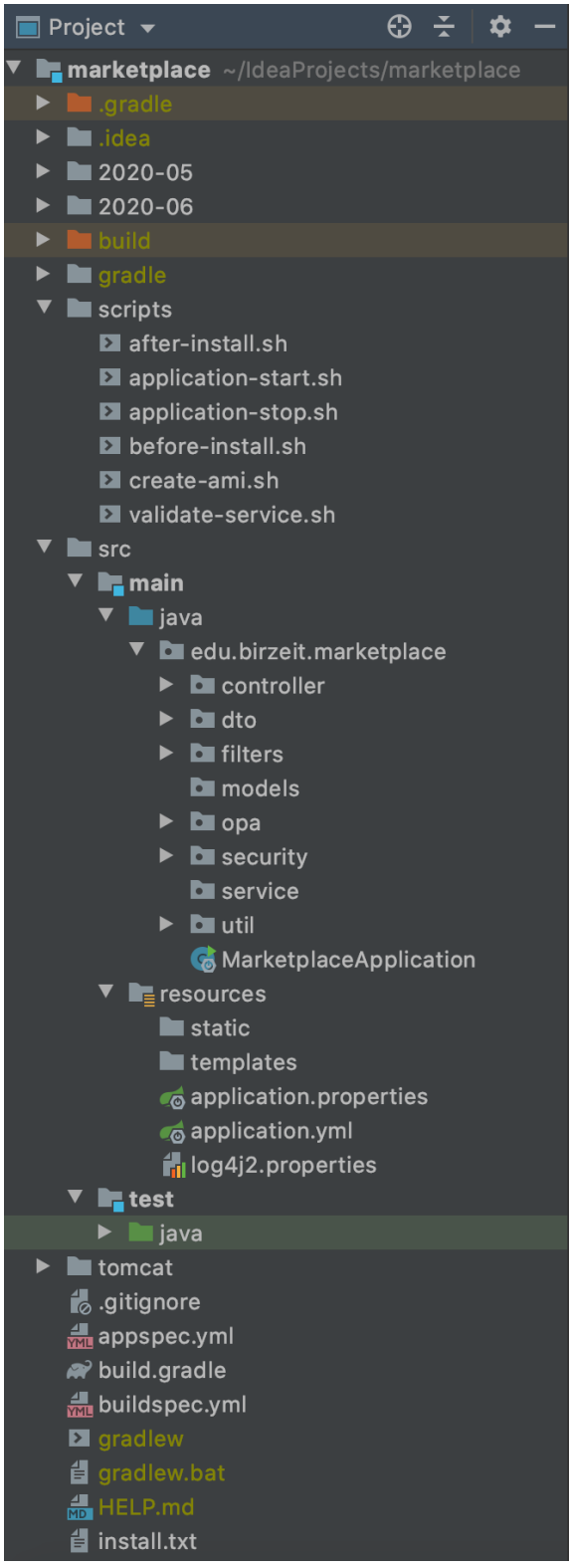

FIGURE 6.1: Microservice Project Structure Sample

6.1.2 API Gateway

We used AWS API Gateway as an external gateway in our security domains. We setup two API Gateways, one for the marketplace security domain and another for the financial security domain. Figure 6.2 shows part of the setup of the API Gateway.

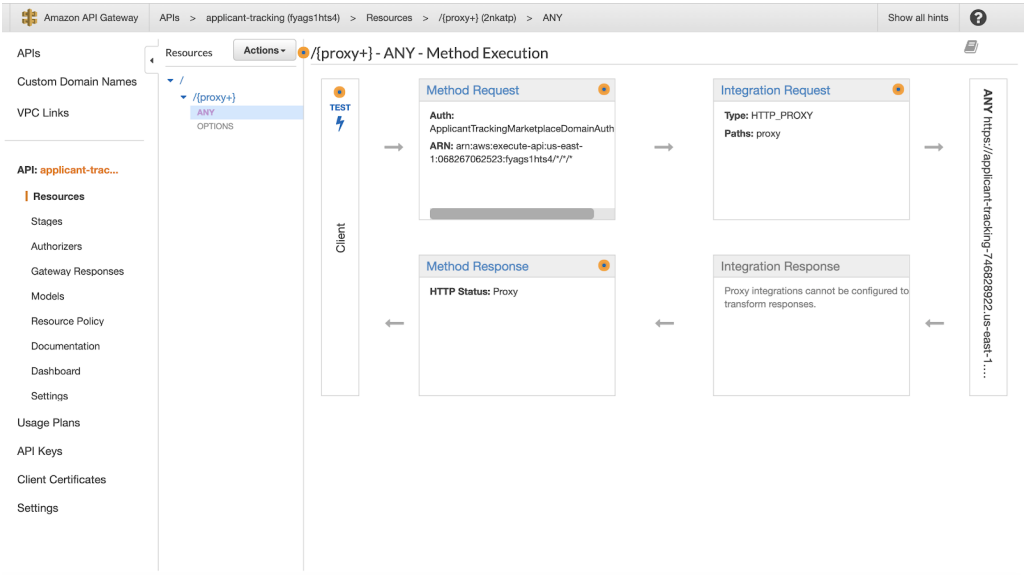

FIGURE 6.2: API Gateway Setup

6.1.3 Authorization Server

We used AWS cognito as an authorization server [11]. We defined two authorization servers, one for each of the security domains. The authorization server is responsible for issuing OAuth2 security tokens in the form of JWT. These tokens are temporal with time to live (TTL) of one hour and have a narrow access that is limited to one security trust domain.

Typically, token expiration is configurable by the authorization server. The smaller the value is, the more frequent the client application needs to refresh

the access token. Threats like replay attacks can impact the system for a longer period of time if the TTL of the security token is set to a large value. When we conducted our experiment back in March, 2020, AWS cognito didn't have the support to customize the token expiration, so we used the default value of one hour for this setting. In August, 2020, AWS announced the support of token expiration in Cognito service, the new settings allows token expiration value to be set between 5 minutes and 24 hours [6]. Figure 6.3 shows part of the authorization server setup.

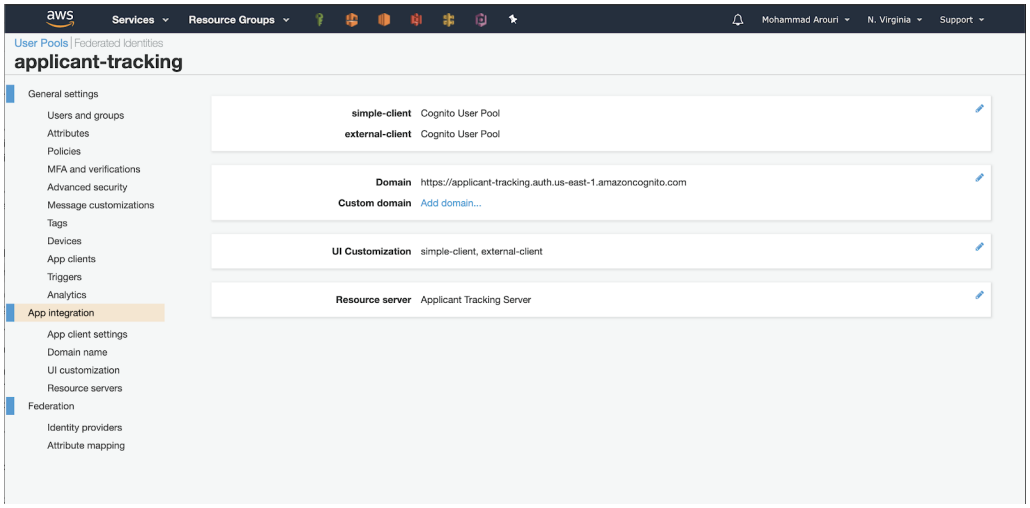

FIGURE 6.3: Authorization Server Setup

In figure 6.4, we show the details of an JWT that was generated by the marketplace authorization server. It contains information about the subject that the token was issued for, and other information about the issuing server. This info will be used later to check the validity of the JWT.

6.1.4 The Sidecar for endpoint Security - Open Policy Agent

Open Policy Agent (OPA) is the authorization framework we used to achieve fine-grained authorization in our security framework. We configured and run

PAYLOAD: DATA


```
{
  "sub": "33c2298b-2863-4b12-8838-6047dce85163",
  "token_use": "access",
  "scope": "edu.birzeit/employer openid",
  "auth_time": 1592072723,
  "iss": "https://cognito-idp.us-east-1.amazonaws.com/us-
east-1_rT2RHbx6Y",
  "exp": 1592076323,
  "iat": 1592072723,
  "version": 2,
  "jti": "80ed63f0-1457-4213-a83c-c9a01697dd59",
  "client_id": "6r9n75m3dune3uok5d8sjaq2sb",
  "username": "employer-1"
}
```


FIGURE 6.4: JWT Sample

OPA version 0.20.5 as a sidecar process. Figure 6.5 shows an authorization policy for the marketplace microservice written in rego (OPA high level declarative language).

6.2 Microservices Security Framework for Authentication and Fine Grained Authorization (MSFAA) in Action

In this section, we demonstrate the appliance of the security framework on the null-architecture and the effectiveness of the security framework in the achievement of its designed security requirements. For demonstration purposes, we use Postman [45], a collaboration tool for API development. Postman plays the role of the client application accessing resource servers on the behalf of a resource owner.

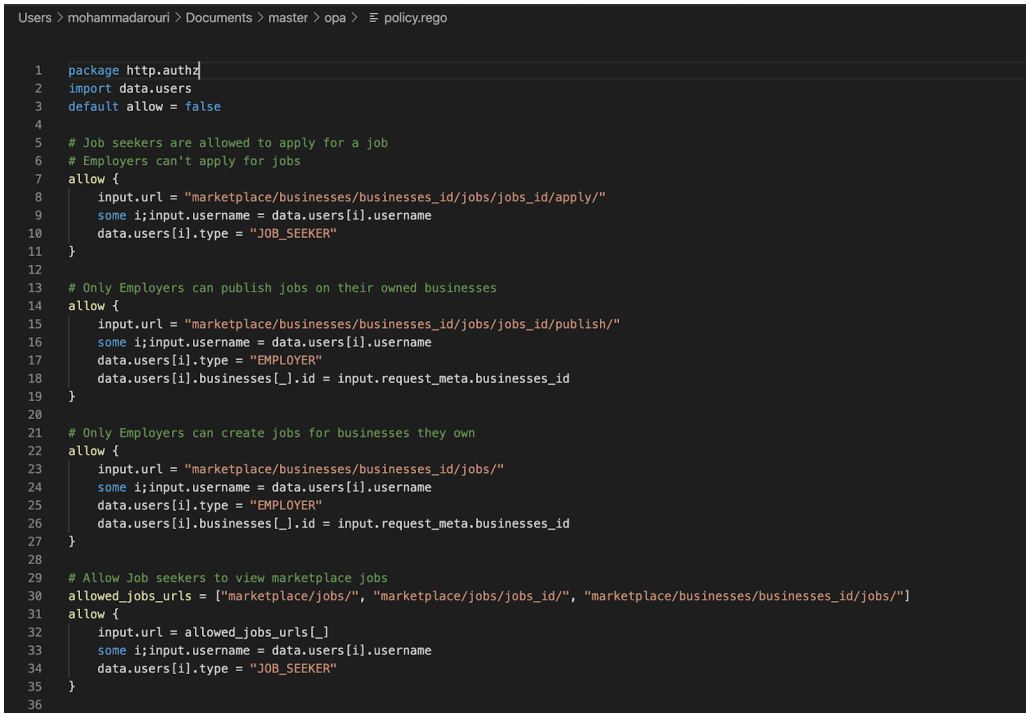

FIGURE 6.5: Open Policy Agent (OPA) Policy Sample Written in Rego

6.2.1 Accessing resources in a single security trust domain

For a client application to be able to successfully retrieve data from a resource server in a security trust domain. It first needs to acquire a valid access token from its authorization server. Figures 6.6, 6.7, 6.8 and 6.9 depict this flow.

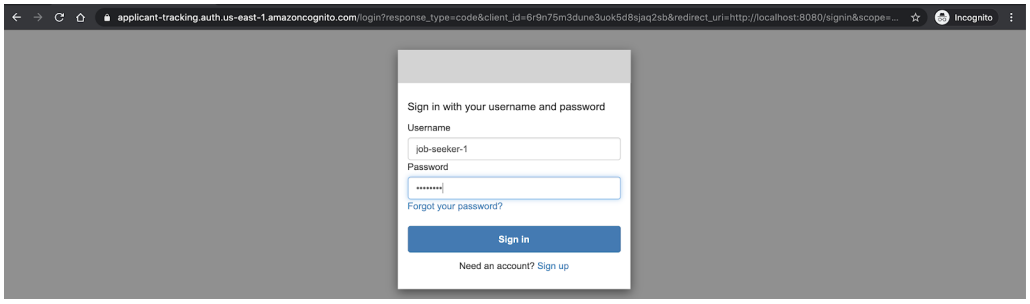

FIGURE 6.6: User Submits Credentials for Authentication

FIGURE 6.7: Authorization Server Redirects Resource Owner to Client App with Authorization Code

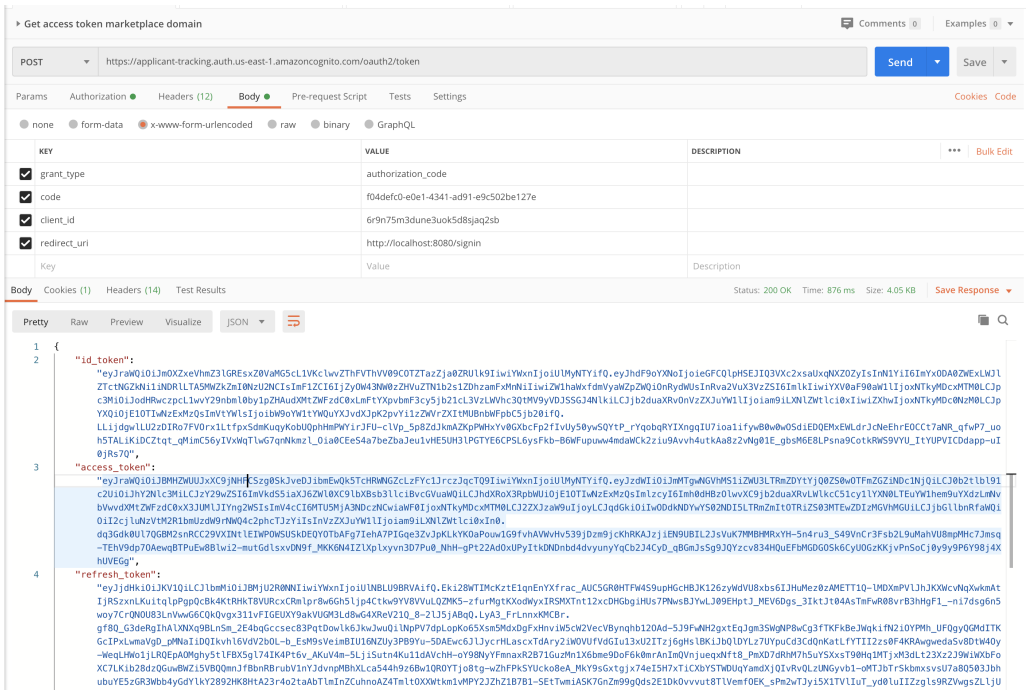

FIGURE 6.8: Client Application Exchanges Authorization Code with an Access Token

After acquiring a valid access token, the client application can call the API Gateway endpoint of the service to retrieve information. Figure 6.10 shows a valid request issued to the API Gateway.

PAYLOAD: DATA


```
{
  "sub": "33c2298b-2863-4b12-8838-6047dce85163",
  "token_use": "access",
  "scope": "edu.birzeit/employer openid",
  "auth_time": 1592072723,
  "iss": "https://cognito-idp.us-east-1.amazonaws.com/us-
east-1_rT2RHbx6Y",
  "exp": 1592076323,
  "iat": 1592072723,
  "version": 2,
  "jti": "80ed63f0-1457-4213-a83c-c9a01697dd59",
  "client_id": "6r9n75m3dune3uok5d8sjaq2sb",
  "username": "employer-1"
}
```


FIGURE 6.9: Access Token Sample Details

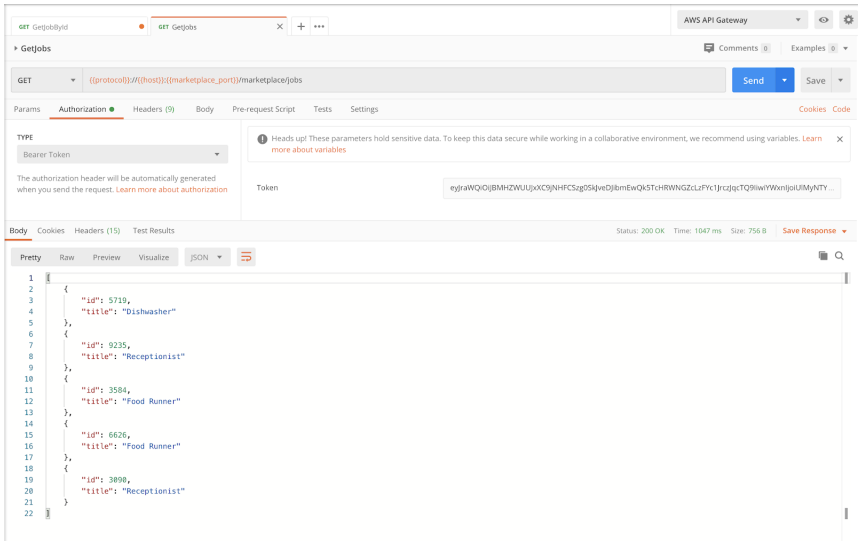

FIGURE 6.10: Authorized API Call Sample

This flow is a direct achievement of the **SecurityRequirement-1** of our security framework; accessing user information needs data owner approval.

Figure 6.11 shows the set of the security checks applied to the API request before it was served by the microservice.

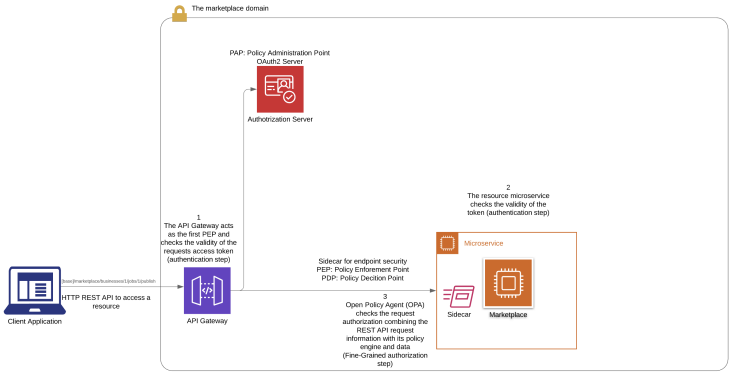

FIGURE 6.11: Security Checks Applied - One Security Trust Domain

Using Open Policy Agent (OPA) framework enabled us to control fine-grained authorization using policies. In this example, only job seekers can view marketplace jobs, employers do not have an access to list and retrieve marketplace jobs to apply for. This is a manifestation for the **SecurityRequirement-1** of our security framework.

Also, OPA runs as a standalone process separating authentication logic from the typical functionalities of the microservice. This is also a manifestation for the **SecurityRequirement-4** of our framework.

If a client application tries to access a service on behalf of a user without proper authentication or with an invalid token. The API Gateway, which is the main central entry point for this type of external call, will reject the request minimizing the architecture's attack surface.

Figure 6.12 depicts this scenario.

In cases of authorization issues, like if an employer tries to get marketplace jobs, API Gateway check will fail to detect this security issue and the call will be passed to the destination microservice. The fine-grained authorization check

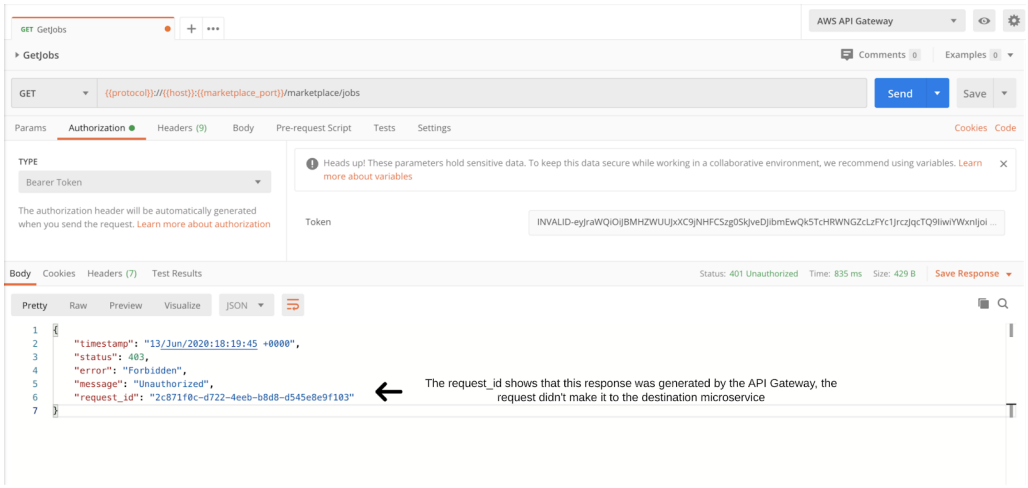

FIGURE 6.12: API Gateway Rejects Unauthorized Request

will process the request and deny the call successfully. Figure 6.13 depicts this flow.

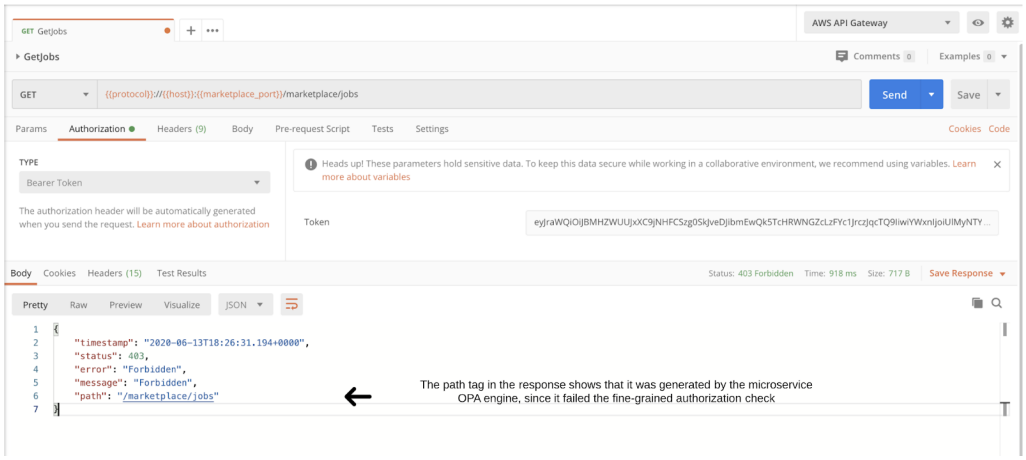

FIGURE 6.13: Open Policy Agent (OPA) Rejects Unauthorized Request

6.2.2 Accessing resources that expand two security trust domains

In the previous section, we demonstrated the effectiveness of the security framework for one security trust domain calls. In this section, we expand that to describe the flows when the API call expands multiple security trust domains.

Figure 6.14 shows the publish job API flow. The API request calls the marketplace microservice (in the marketplace security trust domain) which also needs to check and access the employer's active subscription in the financial microservice that is located in the financial security trust domain.

The first three steps in the flow are just like the typical flow when a single resource is called. The interesting part is in step 4, in order for the marketplace to check the employer's active subscription. It needs a valid access token that is accepted by the financial microservice in the second security trust domain.

In order to do so, the marketplace microservice calls the authorization server in the financial security trust domain and passes it's client id and secret credentials in a standard client credentials OAuth2 grant flow. A more appropriate grant type is to use OAuth2 Token Exchange described in the RFC 8693 [42]. When we started the implementation of our framework, no framework officially supports token exchange grant type. Only keycloak (an open source identity and access management solution managed and maintained by RedHat) have token-exchange grant type in preview not-fully supported mode [28].

Figure 6.15 shows an example for an API that calls the second authorization server in the second security trust domain to acquire a new access token that is accepted by microservices in the second security trust domain.

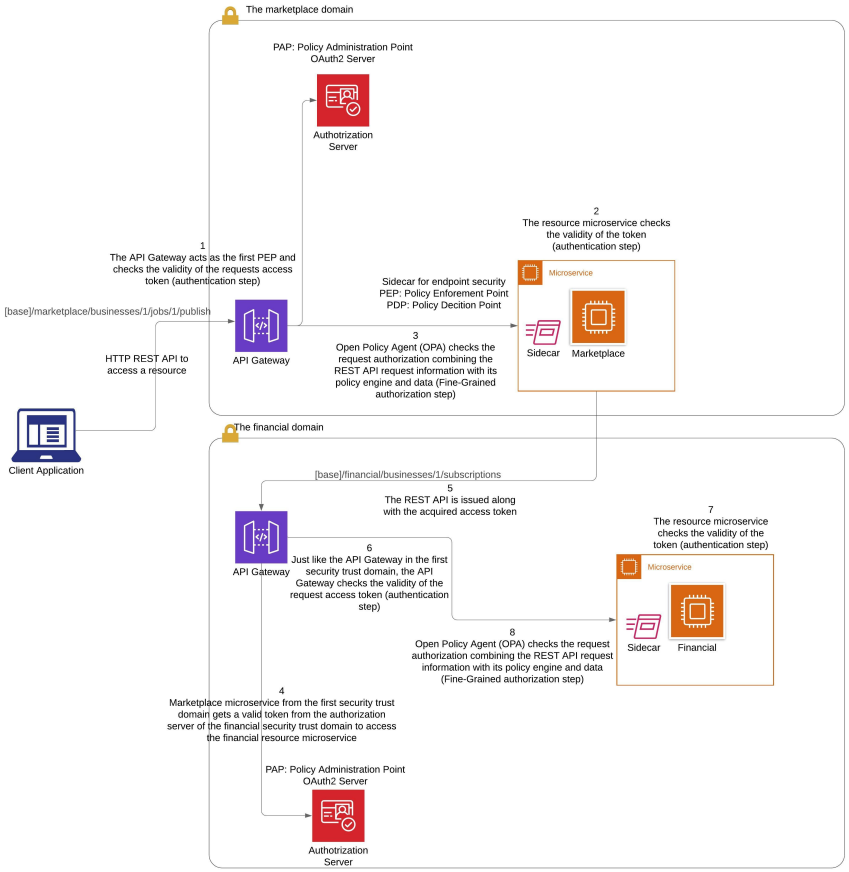

FIGURE 6.14: Multiple Security Trust Domains API Security Checks Flow

Finally, figure 6.16 shows the successful call result. It calls the marketplace microservice which in turn calls the financial microservice to check for active subscriptions.

As shown in this demonstration, the financial microservice protects its data and only allows actors with a predefined access to reach it through the appliance of the authentication and fine-grained authorization security checks. This is a

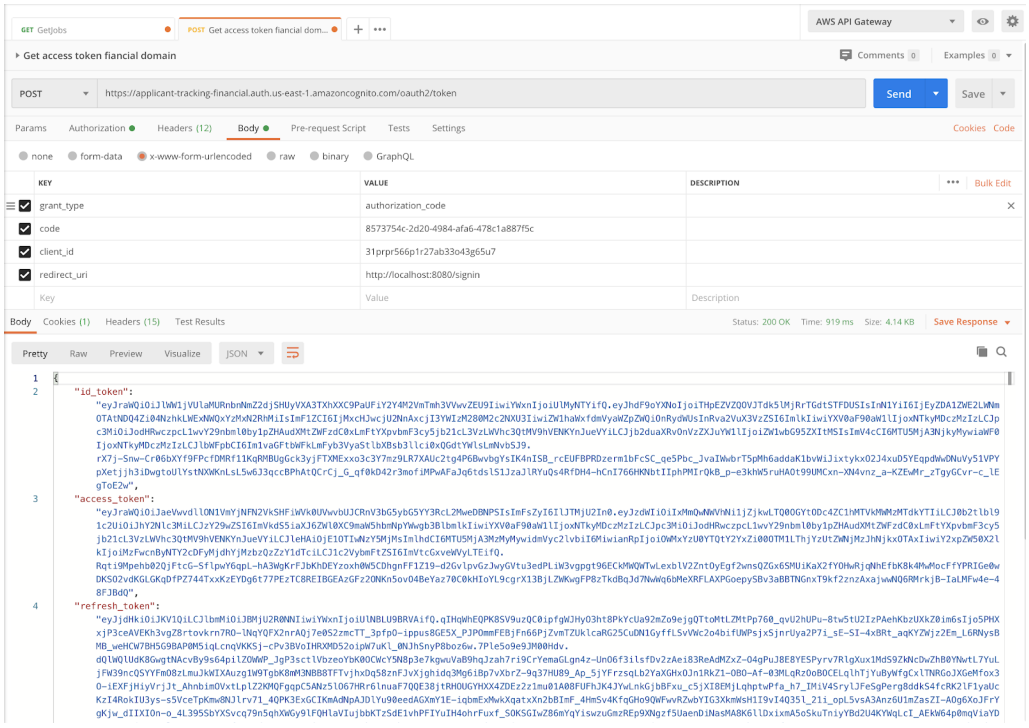

FIGURE 6.15: Acquiring an Access Token for the Second Security Trust Domain

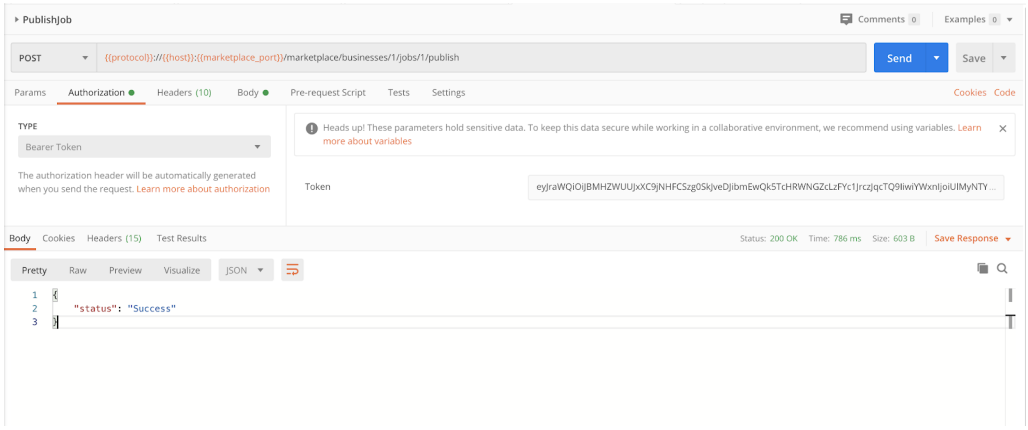

FIGURE 6.16: Authorized API Call Sample - Two Security Domains

manifestation of **SecurityRequirement-3** of the security framework. Another point to mention is that each security token is only valid for the security trust domain in which the token was generated. Accessing the second security trust

domain required us to generate a new access token before calling the financial microservice, no single token has a full access to the entire system. This is an embodiment of the **SecurityRequirement-5** of our security framework. Finally, in case of a successful token theft, the access token will not have full access to the system because the system holds multiple security trust domains. The stolen access token will only be valid in the issuer's security tryst domain. Also, each access token is a temporal token that is valid for a short amount of time (one hour). After that, the token will expire. This is a mitigation for the token theft attack and an implementation of **SecurityRequirement-6** of the security framework. Our choice of one hour was based on AWS Cognito recommended value [49]. This value can be customized based on the application needs.

6.3 Evaluation and Statistics

The experimental design and details was tuned and enhanced over a number of iterations of dry runs. After we made sure that all confounding variables are neutralized, a wet-run was conducted for each of the treatments. In this section, we will show how the experiment was conducted, how the data was gathered, combined and evaluated.

6.3.1 Experiment Runs

In the experiment, we conducted four major runs, one run for each treatment. Table 6.1 shows the basic info of the run results.

TABLE 6.1: Experiment runs results

Run	Samples	Avg Latency	Min Latency	Max Latency	Std. Dev.	Error %	Throughput	Rcvd KB/ sec	Sent KB/ sec	Avg. Bytes
With Security Framework - Two Security Trust Domains	2500	253	223	1055	39.67	0.00%	10.02	5.9	11.7	603
Without Security Framework - Two Security Trust Domains	2500	226	212	1249	27.02	0.00%	10.01	5.06	2.48	517
With Security Framework - One Security Trust Domain	2500	123	110	1136	38.55	0.00%	10.03	5.9	11.76	603
Without Security Framework - One Security Trust Domain	2500	110	105	679	13.72	0.00%	10.03	5.06	2.51	517

The table shows that each run contains 2500 samples. Generally speaking, having more subjects in the experiment will give it a greater statistical power. In the dry run phase, we noticed that having a small number of samples does not represent a stable environment neither for the run and its resources from a side, nor for the results. After increasing the number of samples, we started to see a repeated stable pattern in the output that represents the actual API latency of the system. The main reason we had not more than 2500 samples in each run is the stable results we had. The results indicated that there is no need to increase the sample size.

Figure 6.17 shows a histogram plot that shows the relation between the number of samples and the measured API latency. As the graphs show, most of the API samples lie in a narrow latency margin, this is simply because successfully neutralizing all confounding variables will leave the impact of the security framework overhead to be measured both for single and multiple security trust

domains, which is a stable and semi-consistent overhead.

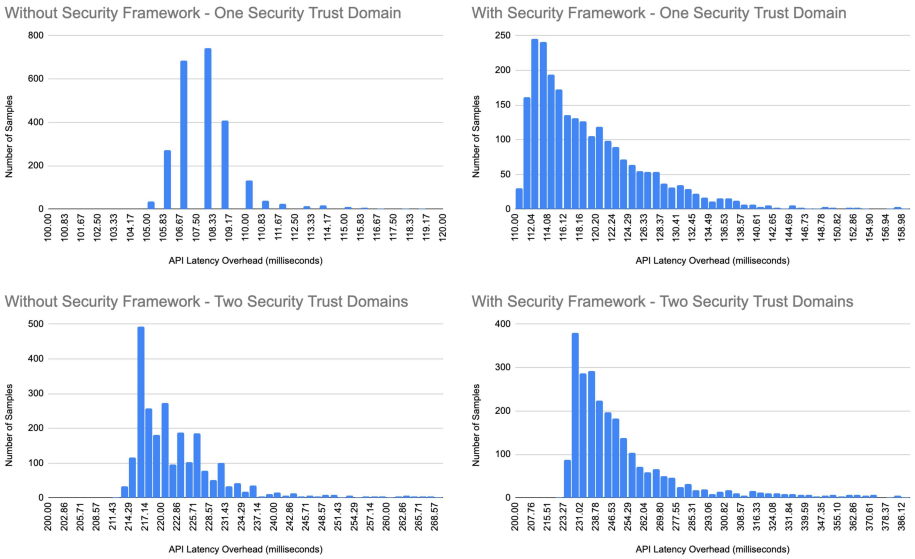

FIGURE 6.17: An Aggregated Overview of API Samples compared to Latency

Figure 6.18 shows the API latency overhead for each individual run over the time. Each one of the four tile represents a unique treatment in our experiment. The x-axis shows the timeline while the y-axis shows the API latency measured in milliseconds.

6.3.2 Normality Tests

After gathering the experiment data, the next step was to check the data normality. We can apply parametric tests on our results if it follows a gaussian distribution. If it is not, we need to follow a non-parametric test. In order to explore our data, we drew a quantile-quantile plot to visually check the data

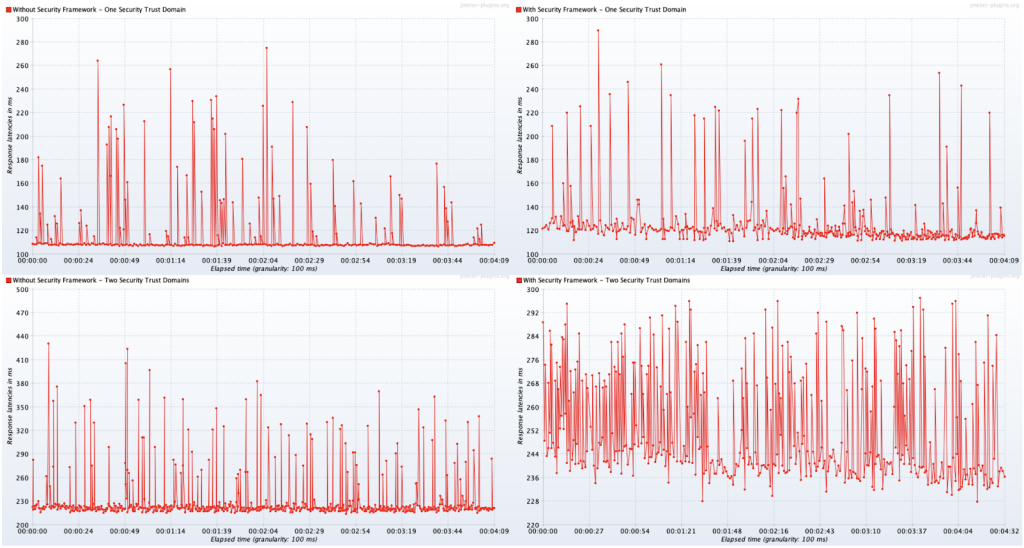

FIGURE 6.18: API Requests Latency

normality. Comparing actual and expected values indicates that the results are not normally distributed. Figure 6.19 shows these results.

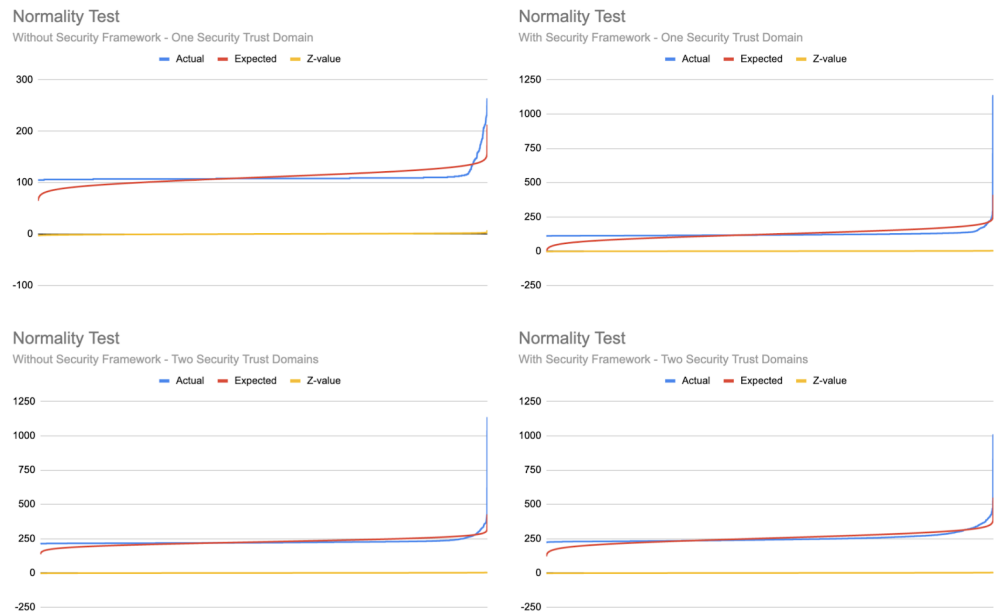

FIGURE 6.19: Quantile - Quantile Plot

To statically validate this finding, we run Shapiro–Wilk normality test [53], the p-value result for all of the four runs were zero. Which clearly indicates that our data is not normally distributed.

6.3.3 API Latency Overhead Statistical Analysis

The next step was to examine the non-parametric tests that suit our case of four groups with a quantitative outcome (the API latency overhead). Kruskal–Wallis was a fit for our experiment. We ran the test on our results and obtained a p-value equals zero. This basically means that we can reject the null-hypothesis and support the alternative hypothesis.

We also applied two Mann-Whitney U tests. Mann-Whitney U test allows two treatments to be compared to each other without the need that their values are normally distributed. We applied a test for the treatments of with and without security framework on a single security trust domain and another test for with and without security frame on two security trust domains. We also obtained a p-value of zero for both tests which also indicates that we should reject our null hypothesis and support the alternative one.

6.4 Summary

In this chapter we discussed the details of our experiment and verified the statistical significance of our results. In the next chapter, we discuss the findings we showed across this chapter, we discuss some aspects that relate to the security framework effectiveness and link these aspects to both the research questions and the current state-of-the-art. We also discuss the results of the experiment and the security framework performance impact on the system.

Chapter 7

Discussion

In chapter 3, we used the current state-of-the-art to address our first research question RQ-1. We identified the main security goals that related to authentication and fine-grained authorization in a microservices architecture. These security goals formed the basis that we used to establish the main security requirements for the proposed security framework.

To evaluate the proposed security framework, we moved into two dimensions. The first one, was to demonstrate the effectiveness of the security framework in a microservices system. While the second one was to measure the performance implications of the proposed framework. We discuss each of these points in the subsequent sections.

7.1 Security Framework Effectiveness

In this section, we discuss the effectiveness of the our proposed security framework from three perspectives:

- Taking advantage of the state-of-the-art security frameworks strengths and limitations.

- Security requirements satisfaction.
- Security framework generalizability and portability.

The following sections discuss the details of these perspectives:

7.1.1 State-of-the-art Security Frameworks Strengths and Limitations

Based on the first research question RQ-1, we were able to identify a set of security frameworks that targets authentication and fine-grained authorization in a microservices architecture. Each of these microservices has its strengths and weaknesses. Yargina's at. el. proposed a security framework that redefined the security perimeter and adopted the defence in depth security principle [62]. Nehme at. el. also used a local API Gateway at the front of each microservice to apply the defence in depth principle [38]. Our security framework took the defence in depth security principle as a first class citizen. We used a sidecar for security endpoint technique to ensure that the security perimeters are pushed inwards toward the microservice boundaries. Using sidecars provided the solution with other advantages including decoupling security from business logic and providing the ability for security policies to be configured, redeployed and managed independently from the microservice logic.

One main difference between our proposed security framework and the existing frameworks is that our security framework took an advantage of the existing system boundary. While defining the microservice boundaries as the security perimeter defence, we added a new centralized authentication check at the API Gateway level. This check will only processes the external HTTP REST calls. This central authentication check provides the system with capability to fail fast when processing illegitimate requests. Being satisfied only by pushing

the perimeter defence toward the microservices makes the microservices themselves the first line of defence and the solo player to protect their resources from outsider attacks. Our proposed check centrally validates every incoming HTTP request and forbids the unauthenticated requests from passing through to the destination microservice.

Yargina's security framework suggested the generation of a new JWT per request [62]. This can be an overwhelming process from a performance perspective. Hitting the authorization server to generate a new token for every API request generates high load on the authorization server and introduces latency overhead to each API. Nehme proposed another approach by using an OAuth client for every pair of microservices [38]. This basically means that the flow will return to the user for notification and consensus in each new microservices integration. This also can be overwhelming especially if the interacted microservices have similar functionalities and lie within the same trust boundaries.

Our proposed security framework suggests dividing the system into multiple security trust domains, where microservices that serve the same business domain and hold related data reside within the same security trust boundary. Each security trust domain has its own authorization server and issues access tokens that are only valid within the boundaries of this particular domain. If a microservice needs to interact with a second microservice outside it's security trust domain. It needs to acquire a new access token from the authorization server in the destination security trust domain.

Following this approach, we gain two main advantages. The first one is avoiding the generation a new access token per API request, which had a positive impact on the performance of our security framework. The second one is

that the usage of multiple security trust domains makes us avoid the usage of powerful tokens, no single access token can have access to the entire system.

7.1.2 Security Requirements Satisfaction

In order to check if the security checks meet the proposed security requirements, we implemented a microservices based-system that represents a real business use case. We also implemented the proposed security framework and applied it to the microservices system. In chapter 6 we showed a set of scenarios to check and demonstrate the satisfaction of the security requirements. We also discussed in detail how each of the security checks contributes to the overall satisfaction of the security requirements.

7.1.3 Security framework generalizability and portability

One of the challenges we faced was to keep the proposed security framework applicable to a broad set of microservices systems. In order to achieve this, the following aspects were taken into consideration during the design of the security framework as well as during the experiment:

- Industrial use case
- Cloud based framework
- Security standards
- Sidecar for endpoint security

Following are the details for each of them:

7.1.3.1 Industrial use case

When we designed our experiment, we picked up a real business use case to form the null-architecture for our experiment. This gave us a better realistic view of the microservices internal details, their architecture and complexities.

The architectural elements that have been used to design the use case are a set of typical components that can be found in any microservices system. This includes the API Gateway and the microservices themselves, no more basic elements are required to be able to apply our security framework. Both API Gateway and microservices components are foundational blocks that can be found in any typical microservices system. The security framework does not require any type of virtualization or containerization layers to exist, it can be applied to both flavors. It can also be applied to microservices systems built on serverless architecture.

7.1.3.2 Cloud based framework

In our experiment, we used AWS cloud provider to build both the null-architecture and the security framework. The International Data Corporation indicated that the majority of the microservices systems are cloud based. By 2021, 80% of application development on cloud platforms will be built using Microservices architecture [31]. This was a motivation to conduct our experiment using one of the well-known public cloud providers.

7.1.3.3 Security standards

Our proposed security framework is built using a set of security standards and frameworks. It uses OAuth 2 for authentication, JWT for security token representation and Open Policy Agent (OPA) for fine-grained authorization. All of

these elements are well known and widely adapted.

The security framework proposed by Nehme [38] uses XACML and the security framework proposed by Preuveneers [46] uses a JSON based lightweight definition of XACML. Despite that XACML is a well known, relatively old and mature standard, it does not see a wide adoption in the industry due to its management complexities. XACML is also not suitable for cloud and distributed deployment[61]. We avoided the use of XACML in our framework and suggested the use of Open Policy Agent (OPA) security framework. It has similar capabilities in terms of fine-grained authorization. It's simpler to use, maintain and manage.

7.1.3.4 Sidecar for endpoint security

We used Open Policy Agent (OPA) as a sidecar for endpoint security, this abstracted the details of the fine-grained authorization engine and decoupled the microservices system from its security framework. Using the sidecar for endpoint security technique allows implementing the proposed security framework not only using Open Policy Agent (OPA), any security framework that is capable of supporting fine-grained authorization can be used.

All of these four aspects helps generalize the security framework applicability to and eases its adaptability in microservices systems.

7.2 Security Framework Performance

7.2.1 Security Framework Performance Implications

In order to answer the third research question RQ-3, we designed an experiment to measure the API latency overhead of the proposed security framework.

Experiment design, setup, applying workload and generating results were discussed in chapter 6.

After running the experiment, we were able to reject the null-hypothesis and support the alternative hypothesis which indicates that there is a difference between the baseline architecture (the null-architecture) and the proposed architecture with security checks applied on the performance of the system (measured by API latency overhead). In order to understand this difference, we drew two graphs. Figure 7.1 shows the API latency overhead comparison between the system without the security framework applied and with the security framework applied given that the system only contains one security trust domain.

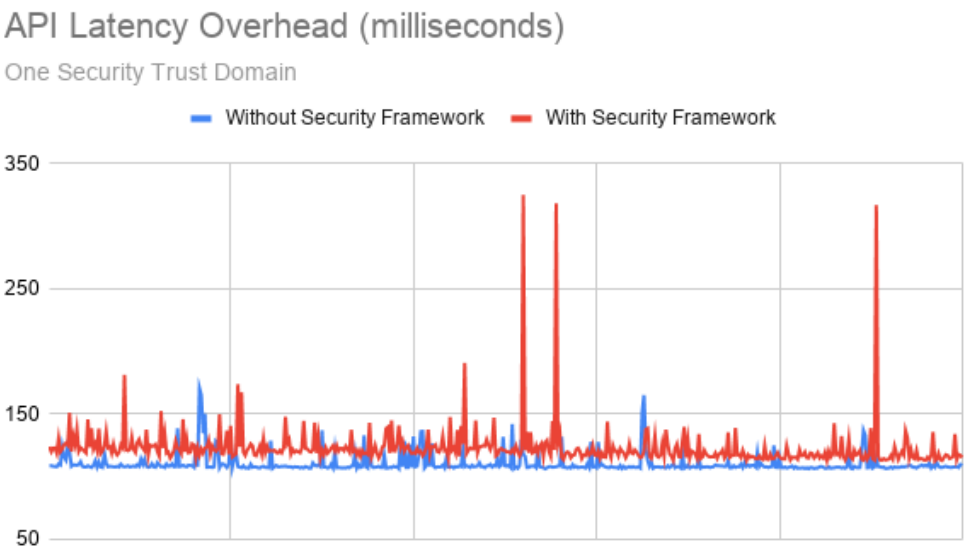

FIGURE 7.1: API Latency Overhead with One Security Trust Domain

The number shows that an overhead of 11.9% results from the addition of the security framework to the system.

The second comparison is shown in figure 7.2. It shows the API latency overhead comparison between the system without the security framework applied and with the security framework applied when the system has two security trust domains.

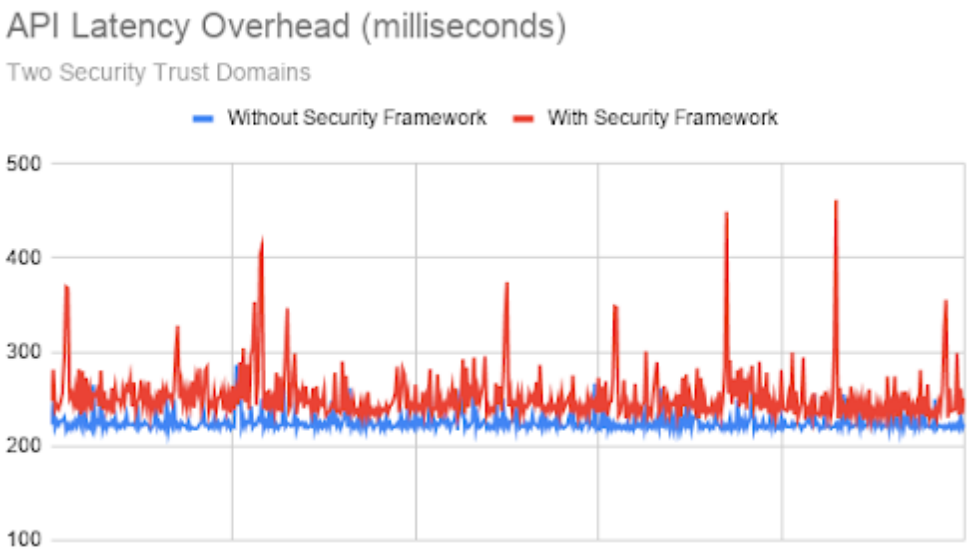

FIGURE 7.2: API Latency Overhead with Two Security Trust Domains

Numbers show that an overhead of 12% results from the addition of the security framework to the system. We noticed that in both cases, the API latency overhead has almost the same percentage; around 12%.

From our experiment perspective, when the system have a single security trust domain with the security framework applied, the total API latency overhead or the Request Trip Time (RTT) can be measured as:

Request Trip Time (RTT) = base request latency overhead + security framework overhead

Where the "base latency overhead" includes all processing phases the request

passes through; including application processing time, API gateway processing time, load balancer processing time and networking time.

When a request is issued to a microservice that can serve the request without the need to call the service of other microservices, the request will pass through one API gateway, one load balancer and will be processed by one microservice application. The existence of the security framework will apply multiple checks on the request at the API gateway stage and at the sidecar stage.

On the other hand, if the request to the destination microservice requires calling other microservices in other security trust domains, additional delays will encounter the request. If the request requires the destination microservice to call a service of only one microservice in a second security trust domain, RTT will almost double assuming that both microservices will take similar time to serve the request. It will pass through two API gateways, two load balancers and two microservice applications.

Same can be said about the security framework checks, it will double as well. The request will be checked by the two API gateways (one for each security trust domain) and by the two sidecars (one for each microservice). The proportion between the number of microservices the request will span across different security trust domains and the number of security checks it will encounter by the security framework is almost linear. We measured this value and it was 12% of the request total trip time. This observation can be generalized if the system has more than two security trust domains.

We believe that this overhead is acceptable in a microservices system due to the nature that microservices are slower than typical monolithic applications.

Microservices require hops between the different services of the system that are separated by the network. We believe that our approach is an acceptable one specially for microservices applications that consider security as an essential and critical aspect.

In their experiment, Yargina stated that their security framework added around 11% overhead to the requests [62]. As we discussed in chapter 3, one of the main differences between Yargina's framework and ours is that their security framework lacks the real existence of a fine-grained authorization support. Nehme discussed that their security framework added 32% overhead in average to each request [38]. They also stated that the overhead of access token checks is minimal compared to the overhead caused by the heavy XACML check.

7.2.2 Implementing a second security framework

During our study, there were three main security frameworks we found in literature which discussed authentication and fine-grained authorization in microservices architecture. We looked into the details for each of these frameworks, their strengths and limitations. We took advantage of the current state-of-the-art and built our security framework taking into consideration not to reinvent the wheel as well as to identify the research gap in a trial to contribute to the current researchers effort, pushing the research in this particular area to cope with the advances in microservices architecture.

We did not implement a proof of concept for Yargine's security framework and compare it with our proposed framework. The main reason was mainly because Yargina's security framework lacks the real existence of fine-grained authorization support [62]. They used mTLS, OAuth2 and JWT to implement the

security framework, without the usage of a fine-grained authorization security framework like XACML or OPA.

The second choice of implementation was using Davy's et. al. security framework [46], the suggested framework handled authentication and fine-grained authorization in a microservices architecture. The authors suggested a lightweight access control policy language that was based on a simplified derived version of XACML, in which they used Java Script Object Notation (JSON) instead of XAML. We didn't see an adoption for this suggestion neither in academia nor in the industry; so we avoided implementing their proof of concept. In our opinion, this is due to two reasons; XACML is relatively old, complex to manage, did not see a wide adoption in the industry and not suitable for cloud and distributed deployment [55]. The second reason is that forked libraries need an active community around them to keep them flexible, agile, maintained and reliable which we didn't find.

The third security framework we focused on was suggested by Nehme et. al. in which they implemented a security framework using OAuth2 and XACML to achieve authentication and fine-grained authorization. They used ForgeRock access management and identity gateway on a local machine where they ran their experiment. We tried to bring this solution to the cloud and apply it to our null-architecture, during the process, we faced a problem where the identity gateway can not be obtained unless we have an active ForgeRock subscription. We tried to apply for the product explaining that we are a group of researchers who are seeking the usage of ForgeRock's products for research purposes but we didn't get an answer.

7.3 Summary

In this chapter, we discussed the security framework effectiveness and its performance implications. In the next chapter, we derive our conclusions, discuss future work and threats to validity.

Chapter 8

Conclusions and Future Work

This chapter discusses the conclusion of the thesis, it also highlights future work. We finish this chapter by discussing internal and external threats to validity.

8.1 Conclusion and Future Work

Microservices architecture is an evolving trend in software engineering that enables building large scale, highly scalable, available and flexible systems. It has been gaining a lot of attention, adoption and momentum in the past few years. This big momentum is pushed by microservices characteristics like enabling technology heterogeneity, using the right tool for the right job, resilience, ease of deployment, better organizational alignment and easier scalability and replaceability.

Microservices are not a silver bullet, their benefits do not come without a price. Microservices have all the inherited complexities of distributed systems, testing, monitoring and security become more challenging and harder to manage. One of the main aspects for microservices security is authentication and authorization.

In this research, we started by exploring microservices authentication and authorization state-of-the-art. We identified the main set of security goals the researchers focused on to address microservices security challenges which helped us highlight the limitations in current practices.

After that, we explored existing authentication and fine-grained security frameworks, their strengths and weaknesses. Some of these frameworks lack the real existence of fine-grained authorization capabilities, others do not use security standards and implement their own security components.

In this thesis, we proposed a new security framework for authentication and fine-grained authorization in a microservices architecture. The framework is built to achieve a set of identified security requirements and mitigates against a set of identified security threats. Our framework is built on top of Open Authorization (OAuth2), JSON Web Tokens (JWT) and Open Policy Agent (OPA). Throughout this research, we discussed the architecture and sequential flows of the proposed security framework.

One aspect that we focused on in this research was to show the effectiveness of the security framework. In order to do so, we built a concrete example of a microservice-based system that is derived from an industrial use case; Applicant Tracking System (ATS). We used this motivating use case as our null-architecture and on top of it, we applied our security framework components. We used this deployment to demonstrate the appliance of the security framework and the achievement of its designed security requirements.

Another main aspect we focused on was the performance implications of the proposed security framework in terms of API latency overhead. In order to

measure that, we designed and conducted an experiment in which we used the previous implementation we built for both the null-architecture and the security framework and on top of that, we applied a workload using Apache JMeter and generated results. We extracted and combined these results to study the performance implications of the proposed security framework. An overhead of 12% is added to the REST API calls that are caused by the security checks which have been added by the security framework.

We believe that this extra performance cost is acceptable in a microservices system due to two reasons. One reason is that microservices are tolerant to delays due to their nature as a distributed system that already uses the network to communicate between services. The second reason is that microservices security is an essential and critical aspect, adding extra performance cost in favor of securing the system is beneficial for the overall system.

Future work that can be developed include:

- Expanding the security framework to include other synchronous communication styles other than HTTP REST APIs, like GraphQL and gRPC Remote Procedure Calls (gRPC) [19, 20].
- Expanding the security framework to include asynchronous communication styles for inter-communication between microservices either in a single or multiple security trust domains.

8.2 Threats to Validity

Internal Validity:

- One of the threats to internal validity of our experiment is the network. During the experiment, we set up all of our resources inside an AWS VPC (Virtual Private Cloud). All communications between microservices, authorization servers, load balancers and JMeter instances happened inside this VPC. Despite that, network effect is a hard factor to eliminate during an experiment and it may affect the accuracy of our API latency measurements.

- JMeter as a testing and sampling tool may have an instrumentation effect in our experiment. To avoid this threat impact, we recalibrated JMeter settings and resources to make sure it has no impact on the experiment result. We also did a number of dry-runs with different sample sizes before the actual wet-run to make sure we have a stable environment and reproducible results.

External Validity:

- In this study, we picked an industrial use case as a microservices system. We tried to pick a use case that represents a wide range of microservices systems. One of the threats is if this use case does not represent the general population of microservices systems. This may impact the generalizability of our study.

Bibliography

[1] Luciano de Aguiar Monteiro et al. "A Survey on Microservice Security–Trends in Architecture, Privacy and Standardization on Cloud Computing Environments". In: International Journal on Advances in Security Volume 11, Number 3 & 4, 2018.

[2] Mohsen Ahmadvand and Amjad Ibrahim. "Requirements reconciliation for scalable and secure microservice (de) composition". In: *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*. IEEE. 2016, pp. 68–73.

[3] Nuha Alshuqayran, Nour Ali, and Roger Evans. "A systematic mapping study in microservice architecture". In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2016, pp. 44–51.

[4] Amazon. *Amazon architecture*. 2019. URL: http://highscalability.com/amazon-architecture.

[5] *Amazon AMD*. URL: https://aws.amazon.com/blogs/aws/now-available-amd-epyc-powered-amazon-ec2-t3a-instances/.

[6] *Amazon Cognito User Pools now supports customization of token expiration*. 2020. URL: https://aws.amazon.com/about-aws/whats-new/

2020/08/amazon-cognito-user-pools-supports-customization-of-token-expiration/.

[7] *Amazon EPS*. URL: https://aws.amazon.com/ebs.

[8] *Amazon Linux 2*. URL: https://aws.amazon.com/amazon-linux-2/.

[9] *amazon web services (aws) - cloud computing services*. URL: https://aws.amazon.com/.

[10] *Apache JMeter™*. URL: http://jmeter.apache.org/.

[11] *AWS Cognito*. URL: https://aws.amazon.com/cognito/.

[12] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices architecture enables devops: Migration to a cloud-native architecture". In: vol. 33. 3. IEEE, 2016, pp. 42–52.

[13] Uber Engineering Blog. *Service-oriented architecture: Scaling the uber engineering codebase as we grow*. 2019. URL: https://eng.uber.com/soa/.

[14] Stefano Calzavara et al. "Testing for integrity flaws in web sessions". In: *European Symposium on Research in Computer Security*. Springer. 2019, pp. 606–624.

[15] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. "Research on architecting microservices: Trends, focus, and potential for industrial adoption". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE. 2017, pp. 21–30.

[16] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[17] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. "Challenges in delivering software in the cloud as microservices". In: vol. 3. 5. IEEE Cloud Computing, 2016, pp. 10–14.

[18] Jean-Philippe Gouigoux and Dalila Tamzalit. "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, pp. 62–65.

[19] *GraphQL*. URL: https://graphql.org/.

[20] *gRPC: A high-performance, open source universal RPC framework*. URL: https://grpc.io/.

[21] D Hardt. *RFC6749-The OAuth 2.0 Authorization Framework. Oct*. 2012. URL: https://tools.%20ietf.%20org/html/rfc6749.

[22] Bret Hartman et al. *Mastering Web services security*. John Wiley & Sons, 2003.

[23] *IETF OAuth Working Group*. URL: www.ietf.org/mailman/listinfo/oauth.

[24] Kasun Indrasiri and Prabath Siriwardena. "Securing Microservices". In: *Microservices for the Enterprise*. Springer, 2018, pp. 347–371.

[25] Kai Jander, Lars Braubach, and Alexander Pokahr. "Defense-in-depth and Role Authentication for Microservice Systems". In: vol. 130. Procedia computer science, 2018, pp. 456–463.

[26] Bokefode Jayant D et al. "Analysis of dac mac rbac access control based models for security". In: vol. 104. 5. International Journal of Computer Applications, 2014, pp. 6–13.

[27] *JSON Web Token (JWT)*. URL: https://tools.ietf.org/html/rfc7519.

[28] *Keycloak Identity and Access Management Solution*. URL: https://www.keycloak.org/.

[29] Joseph Migga Kizza. *Guide to computer network security*. Springer, 2013.

[30] Chirag Langaliya and Rajanikanth Aluvalu. "Enhancing cloud security through access control models: A survey". In: vol. 112. 7. International Journal of Computer Applications, 2015.

[31] Xabier Larrucea et al. "Microservices". In: vol. 35. 3. IEEE Software, 2018, pp. 96–100.

[32] J Lewis and M Fowler. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html.

[33] Steve Lipner. "The trustworthy computing security development lifecycle". In: *20th Annual Computer Security Applications Conference*. IEEE. 2004, pp. 2–13.

[34] *Manage swarm security with public key infrastructure (PKI)*. Jan. 2020. URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki/.

[35] Genc Mazlami, Jürgen Cito, and Philipp Leitner. "Extraction of microservices from monolithic software architectures". In: *2017 IEEE International Conference on Web Services (ICWS)*. IEEE. 2017, pp. 524–531.

[36] *Meaning of granularity in English*. URL: https://dictionary.cambridge.org/dictionary/english/granularity.

[37] Scott Morrison. *Securing Microservice APIs*. O'Reilly Media, Inc., 2018.

[38] Antonio Nehme et al. “Fine-Grained Access Control for Microservices”. In: *International Symposium on Foundations and Practice of Security*. Springer. 2018, pp. 285–300.

[39] Antonio Nehme et al. “Securing Microservices”. In: vol. 21. 1. IT Professional, 2019, pp. 42–49.

[40] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O’Reilly Media, 2015. ISBN: 1491950358. URL: https://www.amazon.com/Building-Microservices-Designing-Fine-Grained-Systems/dp/1491950358?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1491950358.

[41] NGINX. *Microservices at netflix: Lessons for architectural design*. 2019. URL: https://www.nginx.com/blog/microservices-at-netflix-architectural-bestpractices/.

[42] *OAuth 2 Toekn Exchange*. URL: https://tools.ietf.org/html/rfc8693.

[43] *OAuth 2.0*. URL: https://oauth.net/2/.

[44] *Open Policy Agent*. URL: http://www.openpolicyagent.org/.

[45] *Postman*. URL: https://www.postman.com/.

[46] Davy Preuveneers and Wouter Joosen. “Access control with delegated authorization policy evaluation for data-driven microservice workflows”. In: vol. 9. 4. Future Internet, 2017, p. 58.

[47] Chris Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018. ISBN: 1617294543. URL: https://www.amazon.com/Microservices-Patterns-examples-Chris-Richardson/

dp/1617294543?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1617294543.

[48] E Rissanen. *eXtensible Access Control Markup Language (XACML) Version 3.0*. 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[49] Herman Roose. *Cognito*. URL: https://docs.aws.amazon.com/cognito/latest/developerguide/amazon-cognito-user-pools-using-tokens-with-identity-providers.html.

[50] Scott Rose et al. *Zero Trust Architecture*. Tech. rep. National Institute of Standards and Technology, 2019.

[51] Riccardo Scandariato, Kim Wuyts, and Wouter Joosen. "A descriptive study of Microsoft's threat modeling technique". In: vol. 20. 2. Requirements Engineering, 2015, pp. 163–180.

[52] *Secure Production Identity Framework for Everyone*. URL: https://spiffe.io/.

[53] Samuel Sanford Shapiro and Martin B Wilk. "An analysis of variance test for normality (complete samples)". In: vol. 52. 3/4. Biometrika, 1965, pp. 591–611.

[54] Yuqiong Sun, Susanta Nanda, and Trent Jaeger. "Security-as-a-service for microservices-based cloud applications". In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2015, pp. 50–57.

[55] Chalee Thammarat et al. "A secure lightweight protocol for NFC communications with mutual authentication based on limited-use of session

keys". In: *2015 International conference on information networking (ICOIN)*. IEEE. 2015, pp. 133–138.

[56] *The Transport Layer Security (TLS) Protocol Version 1.2*. URL: IETF%20Tools, %20tools.ietf.org/html/rfc5246#section-7.4.6.

[57] Axel Van Lamsweerde. "Goal-oriented requirements engineering: A guided tour". In: *Proceedings fifth ieee international symposium on requirements engineering*. IEEE. 2001, pp. 249–262.

[58] Mario Villamizar et al. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *2015 10th Computing Colombian Conference (10CCC)*. IEEE. 2015, pp. 583–590.

[59] *Why You Can't Talk About Microservices Without Mentioning Netflix*. URL: https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/.

[60] *X.400 Gateway API Specification ; X.400 API Associations*. 1989. URL: https://aws.amazon.com/api-gateway/.

[61] *XACML is Dead*. URL: https://go.forrester.com/blogs/13-05-07-xacml_is_dead/.

[62] Tetiana Yarygina and Anya Helene Bagge. "Overcoming security challenges in microservice architectures". In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE. 2018, pp. 11–20.

Appendices

Appendix A

Architectural and Sequential Diagrams Symbols Definitions

Figure A.1 shows the symbols used across the research sequence and architectural diagrams and a simple description about each of them.

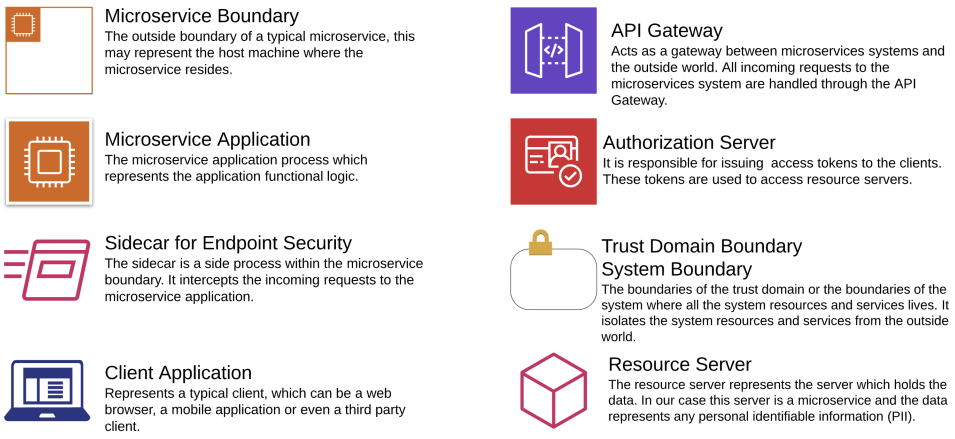

FIGURE A.1: Architectural and Sequential Diagrams Symbol Definitions