Trust in a Distributed Authentication Mesh*

How to create trust and secure communication between distant authentication meshes

Christoph Bühler

Spring and Autumn Semester 2022
University of Applied Science of Eastern Switzerland (OST)

TODO

*I would like to express my appreciation to Mirko Stocker for guiding and reviewing this work. Furthermore, special thanks to Florian Forster, who provided the initial inspiration and technical expertise of the topic.

Contents

List of Figures

Declaration of Authorship

I, Christoph Bühler, declare that this MASTER THESIS titled "Trust in a Distributed Authentication Mesh" and the work presented in it are my own.

I confirm that:

- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this MASTER THESIS is entirely my own work.
- I have acknowledged all main sources of help.
- Where the MASTER THESIS is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Gossau SG, October 10, 2022

Christoph Bühler

1 Introduction

The concept of the "Distributed Authentication Mesh" [1] creates a foundation for dynamic authentication and authorization with diverging authentication schemes. Further, "Common Identities in a Distributed Authentication Mesh" [2] defines and implements the common identity that is transported between services. The mentioned projects show with their respective Proof of Concepts (PoC), that it is possible to authenticate a user and transfer that identity over to other applications that do not share the same authentication mechanism.

However, both projects are only distributed within the same trust zone[1]. While still allowing the "zero trust"[2] principle, the projects do not enable true "distribution".

In the current state, applications within the same trust zone can communicate with each other and a user only needs to enter his credentials (such as username/password) once. When the user is authenticated, the identity (user ID) is encoded in a JWT for other outgoing calls and the receiving party can validate that the user is already authenticated. Then the receiver uses the transmitted information to encode the identity in the corresponding authentication scheme of the destination [1], [2].

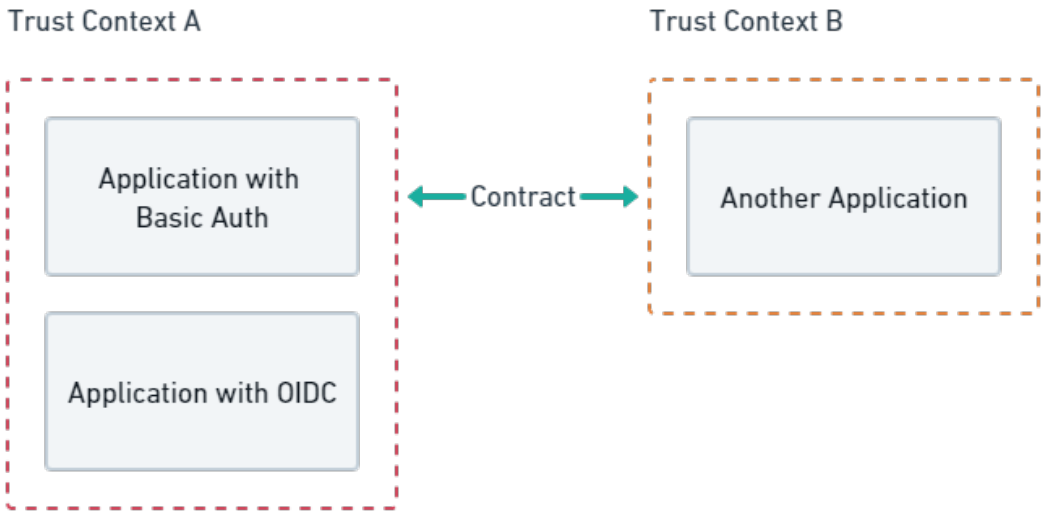

Figure 1: Multiple trust zones that share a contract between them. The contract enables the authentication mesh to verify callers from other zones.

[1] A space where applications can "trust" each other.
[2] Assuming that each call can be compromised, so all presented credentials must be verified for each call.

To achieve true distribution, a contract, as shown in Figure 1, must exist. The contact defines how multiple trust zones can share trust with each other. This project shall define and implement the contract between multiple authentication meshes, such that the Distributed Authentication Mesh can communicate with other trusted zones. To complement the conceptual addition, an open-source implementation of the contract and its components is provided. To demonstrate the contract and the distribution of the authentication, a Proof of Concept (PoC) for Docker is created.

The remainder of this thesis describes prerequisite knowledge, used technologies, and other topics that are required to understand the work. Section 3 shows the current state of the Distributed Authentication Mesh project and which elements are missing for the true distribution between security contexts. The implementation section, Section 4, provides knowledge about the possible technologies for the contract, defines the contract, and implements the contact along with other implementations needed for the working software. The conclusion then gives an overview of the results and provides an outlook into future work.

2 Definitions and Clarification of the Scope

This section provides the scope, context and prerequisite knowledge for this project. It also gives an overview of the used technologies as well as an introduction into the security topic of the project. Note that a deeper introduction into other security related topics is given in the implementation section.

2.1 Scope of this Project

This project builds upon two former projects "Distributed Authentication Mesh" [1] and "Common Identities in a Distributed Authentication Mesh" [2]. The past work defined a general concept for distributed authentication [1] and the definition and implementation of a common identity that is shared between the applications in the mesh [2].

The goal of this project is to achieve a truly distributed mesh. To reach a distributed state in the mesh and to be able to trust other trust zones, a contract between each zone must exist. This project defines and implements the contract and provides the tools that are necessary to run such a mesh in a Proof of Concept. In this project, we analyze different options to form a contract between distant parties and define the specific properties of the contract. After the analyzation and definition, an open-source implementation shall show the feasibility and the usability of the Distributed Authentication Mesh.

Service mesh functionality, such as service discovery even for distant services, is not part of the authentication mesh nor of this project. While the authentication mesh is able to run alongside with a service mesh, it must not interfere with the resolution of the communication. The applications that are part of the mesh must be able to respect the `HTTP_PROXY` and `HTTPS_PROXY` variables, since the Kubernetes Operator of the mesh will inject those variables into the application. This technique allows the mesh to configure a local sidecar as the proxy for the application.

2.2 Introduction into Kubernetes

Since the provided implementation of the Distributed Authentication Mesh runs on Kubernetes, this section gives a brief overview of Kubernetes and the used patterns. Kubernetes is a workload manager that can load balance tasks on several nodes (servers). The explained patterns allow developers to extend the basic Kubernetes functionality.

2.2.1 Basic Terminology

To understand further concepts and Kubernetes in general, some basic terminology and concepts around Kubernetes must be understood.

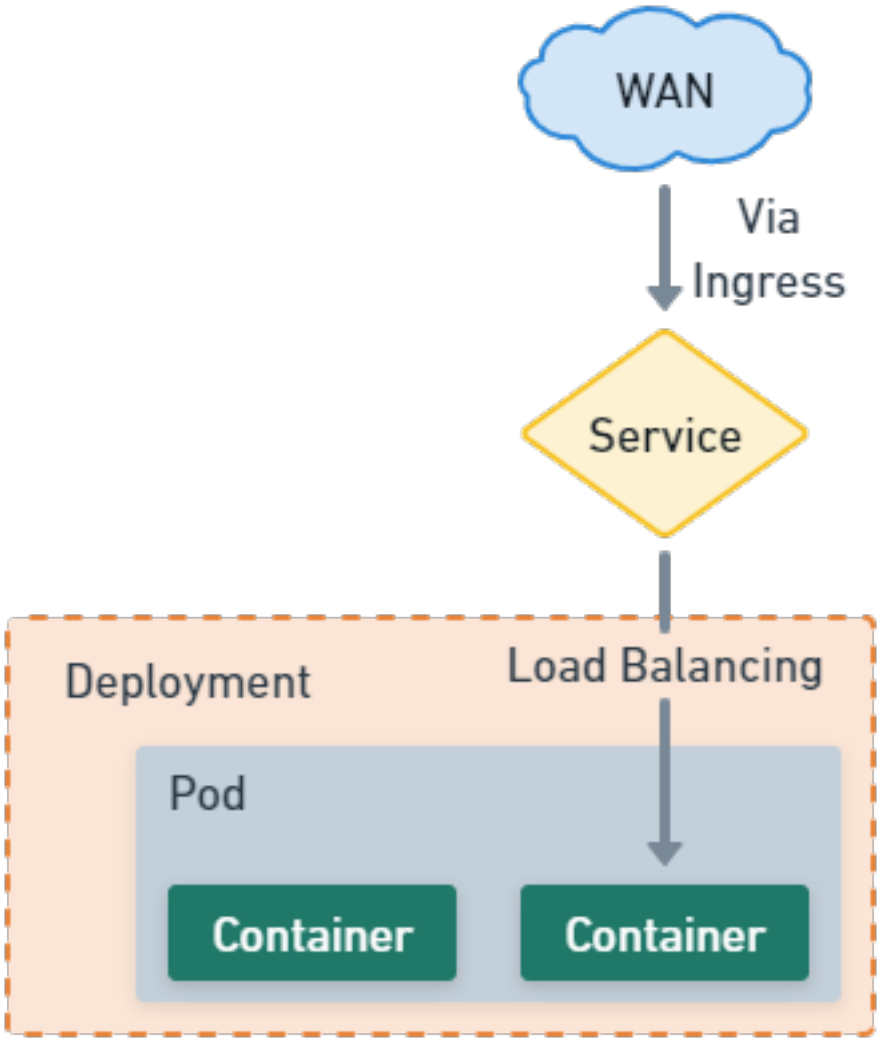

Figure 2: Basic Buildingblocks in Kubernetes

A **Pod** is the smallest possible deployment unit and contains a collection of application containers and volumes [3, Ch. 5]. Figure 2 shows a Pod that contains two containers. Containers are definitions for workloads that must be run. To enable Kubernetes to run such a container, a containerized application and a container image must be present. Such an image-format is "Docker"[3], a container runtime for various platforms.

Deployments manage multiple Pods. A Deployment object manages new releases and

[3]https://www.docker.com/

represent a deployed application. They enable developers to move up to new versions of an application [3, Ch. 10]. In Figure 2, a Deployment contains the Pod which in turn holds containers. There exist multiple deployment specifications, such as `Deployment` and `Stateful Set` which have their own use-cases depending on the specification.

A **Service** makes ports in Pods accessible to the Kubernetes world. They provide service discovery via Kubernetes internal DNS services [3, Ch. 7]. The service in Figure 2 enables access to one of the containers in the Pod. A service load balances access if multiple containers match the service description.

Ingress objects define external access to objects within Kubernetes. Kubernetes uses "Ingress Controllers" that configure the access to services and/or containers [3, Ch. 8]. As an example, "NGINX"[4] is an ingress controller that is popular. When an Ingress is configured to allow access to the service in Figure 2, NGINX is configured that the respective virtual host forwards communication to the given service (reverse-proxying).

2.2.2 What is an Operator

Site Reliability Engineering (SRE) is a specific software engineering technique to automate complex software. A team of experts uses certain practices and principles to run scalable and highly available applications [4]. The "Operator pattern" provides a way to automate complex applications in Kubernetes. An Operator can be compared to a Site Reliability Engineer because the Operator manages and automates complex applications with expert knowledge [5].

An Operator makes use of "Custom Resource Definitions" (CRD) in Kubernetes. These definitions extend the Kubernetes API with custom objects that can be manipulated by a user of the Kubernetes instance [3, Ch. 16]. The Operator "watches" for events regarding objects in Kubernetes. The events can contain the creation, modification, and deletion of such a watched resource. As an example, the "Postgres"[5] database operator reacts to the `Postgres` custom entity. When such an entity is created within Kubernetes, the Operator starts and configures the Postgres database system.

[4] https://www.nginx.com/
[5] https://www.postgresql.org/

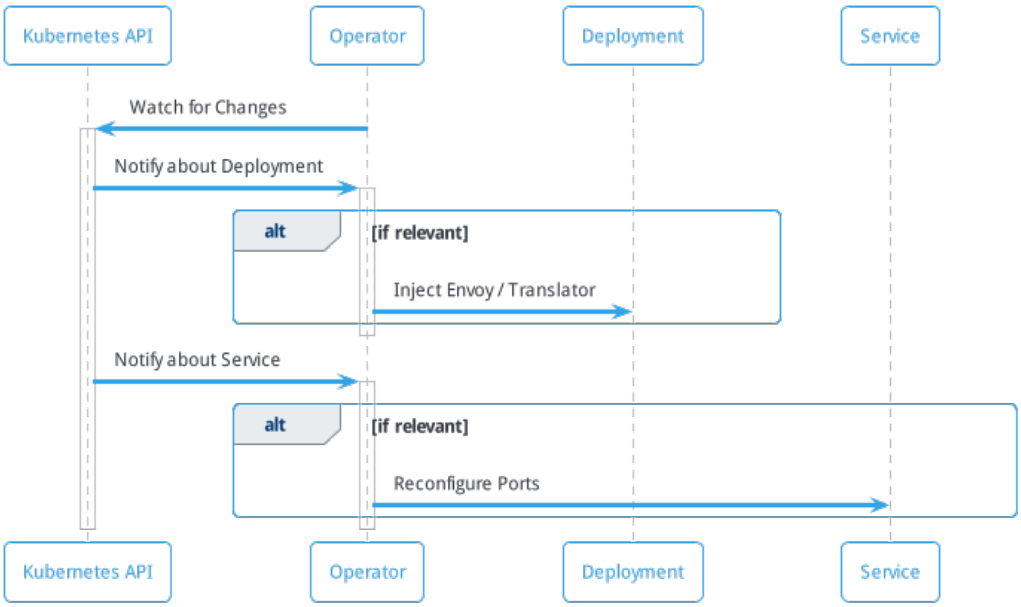

Figure 3: Basic Buildingblocks in Kubernetes

In the Distributed Authentication Mesh, an Operator is used to automatically attach a deployment to the mesh and configure the corresponding services accordingly. As Figure 3 shows, the Operator injects the credential translator and the Envoy[6] proxy into the application (Deployment) and modifies the ports of the service to target the Envoy proxy [1].

2.2.3 What is a Sidecar

A Sidecar is an extension to an existing Pod. Some controller (for example an Operator) can inject a Sidecar into a Pod or the Sidecar gets configured in the Deployment in the first place. [6]

[6]https://www.envoyproxy.io/

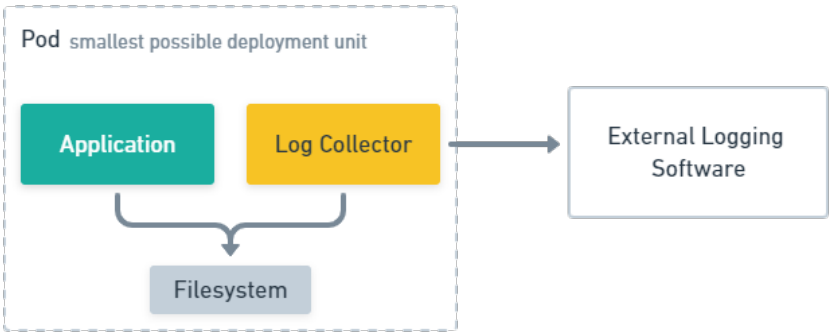

Figure 4: An example of a Sidecar

Figure 4 shows an example of a Sidecar. An application runs a Pod and writes log messages to `/var/logs/app.log` in the shared file system. A specialized "Log Collector" Sidecar can be injected into the Pod and read those log messages. Then the Sidecar forwards the parsed logs to some logging software like Graylog[7].

Sidecars can fulfil multiple use-cases. A service mesh may use Sidecars to provide proxies for service discovery. Logging operators may inject Sidecars into applications to grab and parse logs from applications. Sidecars are a symbiotic extension to an application [3, Ch. 5].

2.3 Security, Trust Zones, and Secure Communication

The Distributed Authentication Mesh is a security application. Therefore, security an important topic in this work. This section gives an overview of the relevant topics to understand further security related concepts. More in-depth knowledge is provided in Section 4.

2.3.1 The CIA Triad

The three pillars of information security: **Confidentiality**, **Integrity**, and **Availability**. These three elements form the foundation of security in information systems. The CIA triad is, despite the fact that it was first mentioned around the year 1980, still relevant for security practitioners and in general security management [7].

Confidentiality addresses the topic of gaining access where one is not allowed to. If someone is able to read certain information without being authorized to do so, the

[7] https://www.graylog.org/

confidentiality is breached. An example could be that some attacker is able to forge login credentials and thus has access to files they should not be able to see.

Integrity covers proving that some information was not modified. When an attacker is able to modify information in a system, even when the attacker is not able to read the information, the integrity of the information is compromised. For example, with a man in the middle (MITM) attack, the integrity of the communication is corrupted and the attacker may forge or change information that the users are sending/receiving [8].

Availability handles the possibility to get the information from the particular system. If an attacker can prevent an authorized user to gain access to their information, the availability is impaired. This could happen, if an attacker uses a DDoS (distributed denial of service) attack to prevent access to a resource.

2.3.2 Trust Zones and Zero Trust

Trust zones are the areas where applications "can trust each other". When an application verifies the presented credentials of a user and allows a request, it may access other resources (such as APIs) on the users' behalf. When the concept of trust zones is applied, other APIs may trust the original requester that the user has authenticated itself. Typically, this is used in microservice architectures where only one point of access (the gateway into the zone) is exposed to the outside world. The APIs behind the application can then share the trust that the gateway created.

In contrast to trust zones, "Zero Trust" is a security model that focuses on protecting (sensitive) data [9]. Zero trust assumes that every call could be intercepted by an attacker. Thus, for the concept of zero trust, it is irrelevant if the application resides in an enterprise network or if it is publicly accessible. As a consequence of zero trust, user credentials must be presented and validated for each access to a resource [10].

2.3.3 Securing Communication between Parties

The key focus of the Distributed Authentication Mesh is the possibility to provide a secured identity over a service landscape that has heterogeneous authentication schemes [1]. Thus, securing communication between participants is of most utter importance. A wide range of security mechanisms and authentication schemes exist. To demonstrate the Distributed Authentication Mesh and the contracts between the trust zones, the following schemes/techniques are used.

2.3.3.1 HTTP Basic Authentication

The "Basic" authentication scheme is defined in **RFC7617**. `Basic` is a trivial authentication scheme which provides an extremely low security when used without HTTPS. Even with HTTPS, Basic Authentication does not provide solid security for applications. It does not use any real form of encryption, nor can any party validate the source of the data. To transmit basic credentials, the username and the password are combined with a colon (`:`) and then encoded with Base64. The encoded result is transmitted via the HTTP header `Authorization` and the prefix `Basic` [11].

2.3.3.2 OpenID Connect

OpenID Connect (OIDC) is not defined in an RFC. The specification is provided by the OpenID Foundation (OIDF). OIDC extends OAuth, which is defined by **RFC6749**. The OAuth framework only defines the authorization part and how access is granted to data and applications. OAuth does not define how the credentials are transmitted [12].

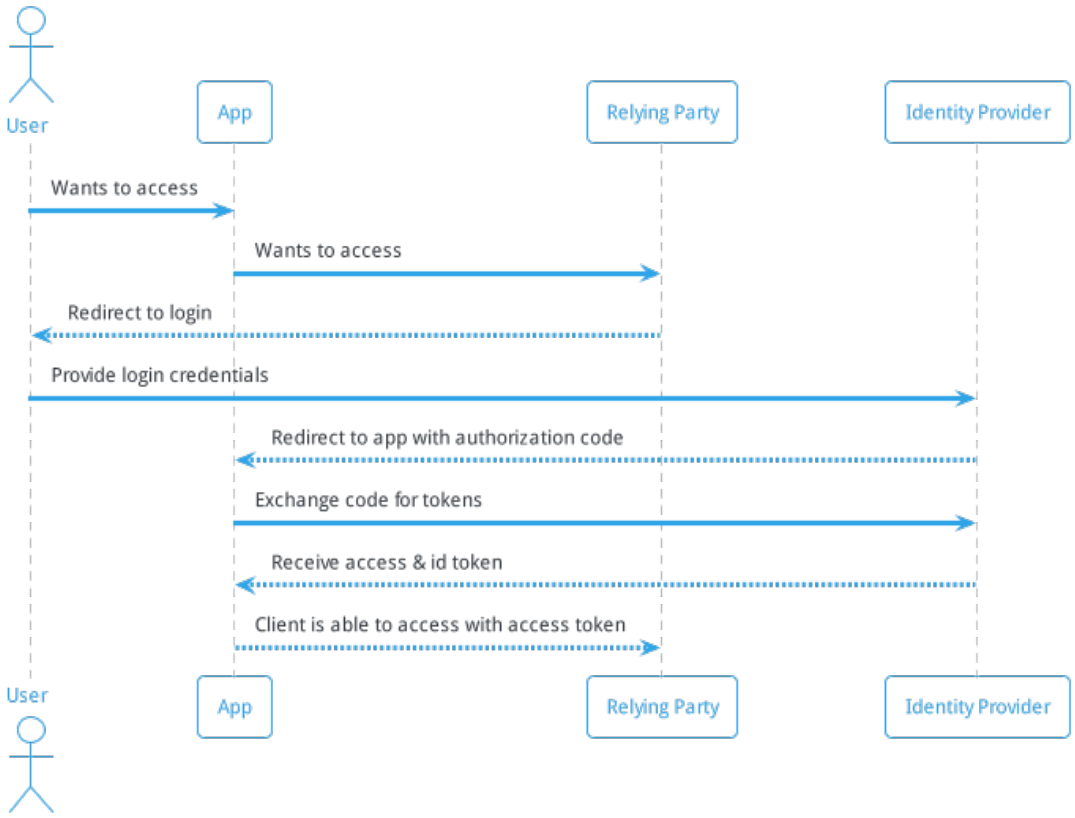

Figure 5: OIDC code authorization flow [13]. Only contains the credential flow, without the explicit OAuth part. OAuth handles the authorization whereas OIDC handles the authentication.

Figure 5 shows an example where a user wants to access a protected application. The user is forwarded to an external login page (Identity Provider) and enters his credentials. When they are correct, the user gets redirected to the web application with an authorization code. The code is used to fetch an access and ID token for the user. These tokens identify, authenticate and authorize the user. The application is now able to provide the access token to the API (Relying Party). The API itself is able to verify the presented token to validate and authorize the user.

2.3.3.3 Mutual Transport Layer Security (mTLS)

An mTLS connection is essentially a TLS connection, like in HTTPS requests, but both parties present an X509 certificate. The connection is only allowed to open if both parties present a valid and trusted certificate. Thus, it enables both parties to verify their corresponding partner and prevents man in the middle attacks [14].

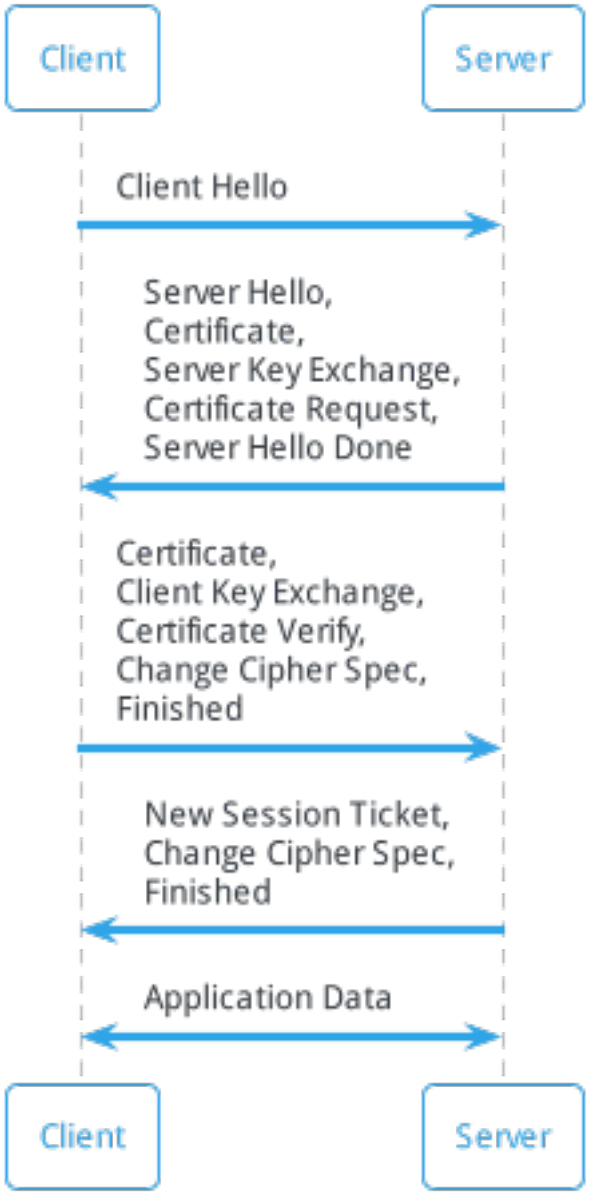

Figure 6: The mTLS Handshake for Client and Server

To establish an mTLS connection, the TLS handshake defined in **RFC5246** is used. Figure 6 shows such a handshake. The client sends its `Client Hello` and is then greeted by the server with a `Server Hello` response. The server response also includes the servers certificate to authenticate itself. The server requests a certificate from the client in the same response. The client can then verify the certificate of the server and returns its own certificate with other TLS related messages. When both parties verify the identity

of the other party, the handshake is completed and the connection is established [15]. The biggest advantage of mTLS is that both parties can verify the identity of the other party. Thus, it is not possible to impersonate a client or a server.

3 The State of Distributed Authentication

This section shows the current state of the art of the Distributed Authentication Mesh. Further, it describes the deficiencies that this project solves.

3.1 Distributed Authentication Mesh within a Trust Zone

The concept "Distributed Authentication Mesh", as described in [1], allows applications to communicate with each other, even if they do not share the same authentication schemes.

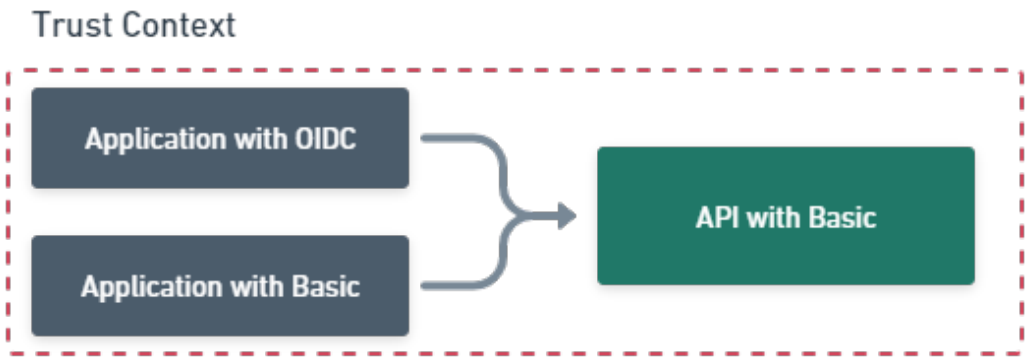

Figure 7: Two applications can communicate with an API, despite the fact, that the API only supports HTTP Basic authentication. The possibility to access an API with diverging authentication schemes is the basic principle of the Distributed Authentication Mesh [1].

Figure 7 shows the concept of the "Distributed Authentication Mesh". Both applications can communicate with the API, but they do not necessarily share the same authentication and authorization mechanisms. The mesh provides the means to translate authentication information into a common identity and transmit it to the receiving application. There, the common identity is translated back into the required authentication credentials (an HTTP Basic authorization header for example) [1].

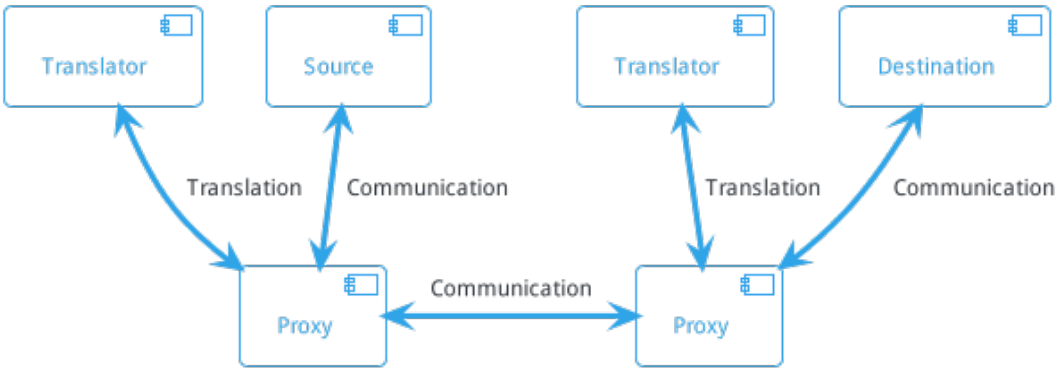

Figure 8: Network Architecture in the Distributed Authentication Mesh

The "Distributed Authentication Mesh" builds upon the idea, that a proxy acts as mediator between source and destination. This proxy then uses an external service, the "Translator". The translator receives incoming and outgoing calls and has the ability to modify the requests. However, it must not interfere with the data plane. It shall only modify HTTP headers and allow or reject a connection. The translator is able to convert the provided authentication information (if any) into a common format.

The common identity is defined as a simple user ID. The ID is encapsulated into a JSON Web Token (JWT) and then signed by the client certificate that the translator receives from the PKI (public key infrastructure). The JWT is then sent to the destination application where the JWT is parsed and validated. The ID is extracted from the JWT and the information can be translated into the corresponding authentication credentials (for example, username/password combination for HTTP Basic) [2].

3.2 Multiple Trust Zones and Distribution

In its current state, the Distributed Authentication Mesh is able to run inside the same trust zone with a shared common identity [1], [2]. The mesh handles the conversion of authentication information (such as an access token or a login/password combination) by transforming it into a shared format. A sender encodes the user ID in a JWT and signs it with its own private key. The receiver can then verify that the information is not modified and that the sender is part of the authentication mesh.

However, the connection between the participants is prone to attacks in multiple ways. The concept only works, if all applications of the mesh are within the same trust zone (for example in the same Kubernetes cluster behind the same API gateway). If part of the application runs on a different cluster, the same trust cannot be applied. An attacker may get their own key material from a mesh PKI and can pose as a valid participant of

the mesh. Therefore, the confidentiality and integrity are violated. Further, the receiving end of the communication has no possibility to verify the sender of the message for certain.

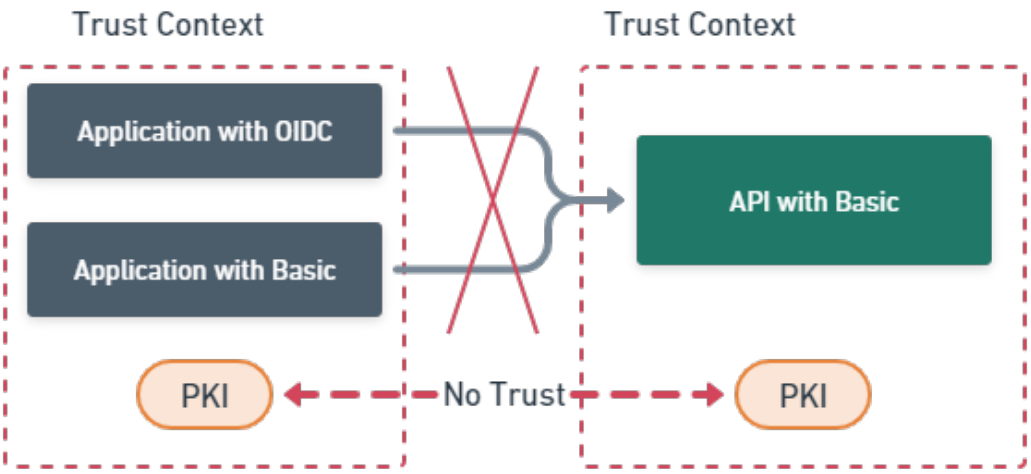

Figure 9: Distributed Authentication Mesh with Multiple Trust Zones

The situation in Figure 9 shows the basic problem of the "Distributed Authentication Mesh". It is not truly distributed over multiple clusters and trust zones. It can only be used within a single trust zone, as Figure 7 showed. The communication between the application and the API could be intercepted by an attacker. An attacker could fetch its own key material from either PKI and then pose as a valid member of the mesh since the common identity only stores the user ID in the JWT [2].

3.3 Contracts for Distribution

To achieve true distribution in the authentication mesh, the mesh needs a possibility to form trust between different trust zones. Various trust zones must establish contracts between them that function as a trust anchor. Trusting another "zone" shall result in an exchange of the public keys of their respective PKIs. With that contract, the mesh can allow its participants to use mutual TLS (mTLS) instead of normal HTTP connections. When mTLS is in place, sender and receiver of the communication can verify that they communicate with the correct entity and thus can verify if a trust anchor between the two exists.

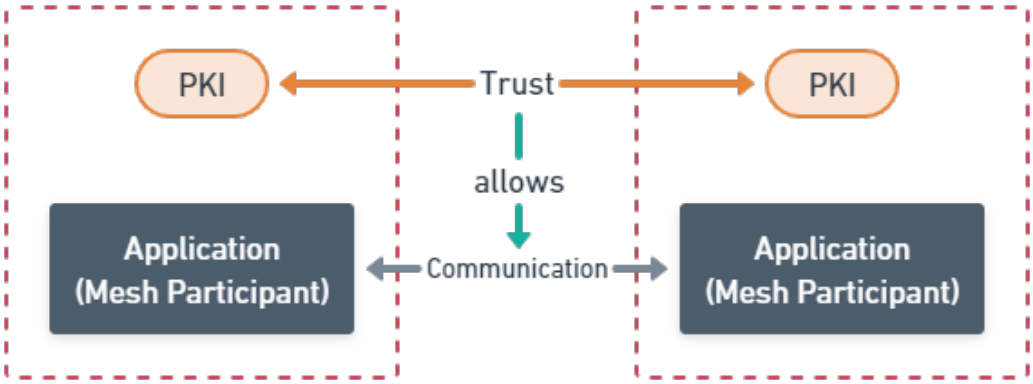

Figure 10: Creating Trust with a Contract

Regarding Figure 10, a contract between the two trust zones creates the trust anchor between the zones. This trust further allows the mTLS connection between the applications to be established. If the connection can be created (i.e. it is not rejected by either side) the participants trust each other and are who they pretend to be.

4 Creating a Trust Context for the Authentication Mesh

This section gives an overview of the created demo applications, the programming language Rust, and security topics that are relevant for the implementation of the authentication mesh. Furthermore, the section describes the implementation of the trust contract and the relation to the authentication mesh.

4.1 Demo Applications

To demonstrate and test the implementation of the trust context and the mesh, multiple demo applications are used. All applications are hosted on GitHub in the open-source repository https://github.com/WirePact/demo-applications. There exist six different applications that are described below.

The **basic_auth_api** is a simple API application written in Go[8]. It uses HTTP Basic Authentication (RFC7617) to authenticate calls against its endpoints. The API can be configured with three different environment variables (`PORT`, `AUTH_USERNAME`, and `AUTH_PASSWORD`). An HTTP web framework package "Gin" provides the HTTP middleware for Go.

```
router := gin.Default()
secure := router.Group("/", gin.BasicAuth(gin.Accounts{
    config.Username: config.Password,
}))
secure.GET("swapi/people", getPeopleFromSwapi)
router.OPTIONS("/swapi/people", cors)
```

The code above shows the implementation of the HTTP Basic Authentication in the Go application. The `gin.BasicAuth` function is used to create a middleware that is applied to the `secure` group. The middleware checks the HTTP request for the `Authorization` header and validates the credentials against the given accounts. The named map `gin.Accounts` is a map that contains username / password combinations. The `getPeopleFromSwapi` function is called if the authentication was successful.

The static website **basic_auth_app** provides a trivial way of accessing any basic protected API. The site runs within an NGINX and contains minimal code. Since this site is hosted statically and does not call API endpoints through some backend logic, it is not possible to adhere to the `HTTP(S)_PROXY` environment variable to route traffic through a specific proxy.

In contrast to the basic auth app, the **basic_auth_backend_app** is an `ASP.NET` application that also uses the HTTP Basic mechanism to authenticate requests. However, the application runs in an ASP.NET context. Thus, it is possible to respect the `HTTP_PROXY`

[8] https://go.dev/

and `HTTPS_PROXY` variable and route traffic through a specific proxy. The application shows a trivial GUI in which the user can specify an API endpoint and a username / password combination.

To provide a more complex authentication scheme, the **oidc_api** authenticates requests against its API via `OAuth2.0`. When the API receives an access token from a client, it uses token introspection (defined by **RFC7662**) to validate the token and authenticate the user [16]. The API needs an issuer, a client ID, and a client secret to validate the given tokens.

```
builder.Services
    .AddAuthentication("token")
    .AddOAuth2Introspection("token", o =>
    {
        var section = builder.Configuration.GetSection("Oidc");
        o.Authority = section.GetValue<string>("Issuer");
        o.ClientId = section.GetValue<string>("ClientId");
        o.ClientSecret = section.GetValue<string>("ClientSecret");
        o.DiscoveryPolicy = new()
        {
            RequireHttps = false,
            ValidateEndpoints = false,
            ValidateIssuerName = false,
            RequireKeySet = false,
        };
    });
```

The code above shows the configuration of the C# API application. It enables the API to verify an incoming access token by using the introspection endpoint of the OIDC provider. The introspection endpoint is defined in **RFC7662** [16].

To complement the OIDC API, an **oidc_app** provides the means to access an OIDC (OAuth2.0) protected API via an application. This Next.js application authenticates users against the OIDC provider and then renders a simple page. Since this is a hosted application, the `HTTP(S)_PROXY` variable is respected. The app calls the API and attaches the access token in the `HTTP Authorization` header. The API validates the token and returns the requested data or denies the request.

The final demo application is the **oidc_provider**. It is based on a Node.js package that provides OIDC server capabilities. This identity provider allows any user with any password and thus is not suitable for production environments. The provider supports OAuth 2.0 Token Exchange (**RFC8693**) to enable the proxy applications to fetch an access token for a specific user [17].

4.2 The Rust Programming Language

To achieve the goals of this work, the programming language "Rust" provides a solid base to implement the contract repository and other system relevant parts. Rust itself is a multi-paradigm language that supports object-oriented features as well as functional components. Rust further allows low-level memory management without the need for garbage collection. Despite the absence of garbage collection, Rust guarantees memory safety. To achieve it, Rust uses a special type checking mechanism that allows the compiler to calculate the lifetime of references and the ownership of the data [18].

Since the compiler of Rust ensures that data can only be modified once and that code has no side effects, the language enables developers to create reliable and secure software. The strict compiler and the vast speed of the compiled results were the primary reasons for choosing Rust as the programming language for this work. The Rust language has comparable performance to C and C++ and is therefore suitable for fast reacting systems like the authentication mesh [19].

With the calculation of ownership and the transfer of ownership, Rust ensures that data can only ever be manipulated by one instance (its owner). No object can be modified without specifically taking ownership. Even though Rust allows an `unsafe` keyword, the code that it contains must be safe and is checked like normal Rust code. This was proven by Ralf Jung et al. by giving a formal safety proof for the language (and the `unsafe` parts in its standard library) [20].

To demonstrate the advantages of Rust and its compiler, consider the following code examples taken from the article "Safe Systems Programming in Rust" [21]:

```
std::vector<int> vec {10, 11};
// Create a pointer into the vector.
int *vectorPointer = &vec[1];
v.push_back(12);

// Bug ("use-after-free")
std::cout << *vectorPointer;
```

The C++ code above creates a vector of integers with two initial elements. Next, a pointer to the second element in the growable array is created. When the new content (`12`) is added to the vector, the backing memory buffer may be reallocated to allow the new object to be stored. The pointer now still points to the old memory address and therefore is a "dangling pointer" [21].

```
let mut vec = vec![10, 11];
let vector_pointer = &mut vec[1];
vec.push(12);
```

```
// This creates a compile error, since the vector is moved.
println!("{}", *vector_pointer);
```

The Rust compiler does check usage of data and references statically and therefore does not allow the use of a dangling pointer. The compiler will give the following error message for the code above: “cannot borrow vec as mutable more than once at a time.” [21].

During this project, all existing elements of the Distributed Authentication Mesh were rewritten to the Rust programming language. Since the communication between the moving parts of the system use gRPC to communicate, the framework or language behind the system does not really matter.

4.3 Sign and Distribute Contracts between Participants

This section shows how a contract between two parts of the authentication mesh can be created and distributed. To enable the authentication mesh to be truly distributed, the PKI of each trust zone must have a contract to create trust between them. Since each PKI creates its own root certificate, other PKIs must be able to verify and trust the root CA of other PKIs.

4.3.1 Using a Blockchain

One possibility to create and share such contracts is the usage of Blockchain. Blockchain and smart contracts allow participants to validate the transaction history of the chain and therefore give a possibility to create trust between the parties.

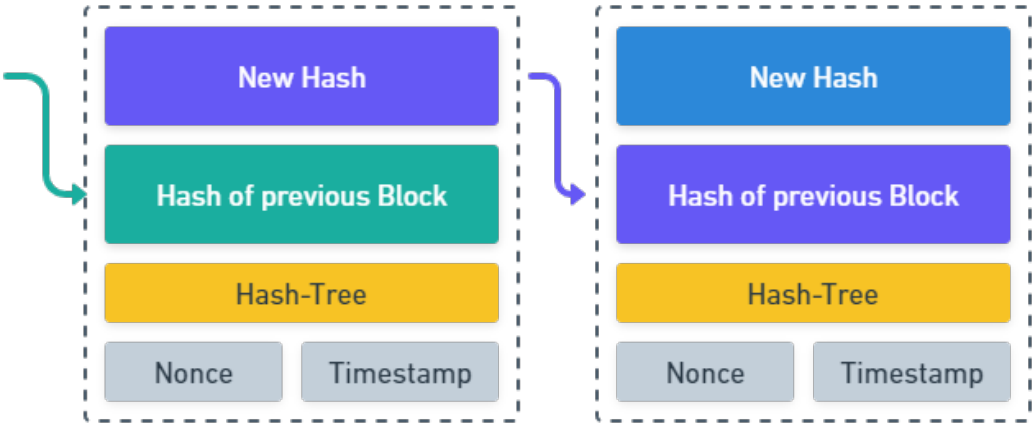

Figure 11: Basic Principle of a Blockchain

4.3.1.1 Introduction into Blockchain The basic principle, stated in Figure 11, shows how new blocks in the chain come to existence. The first block is called the "genesis block" and has no information about any previous blocks. All blocks down the chain contain information about the previous block. Along with the previous hash, each block contains a hashed history of all transactions [22].

The transaction history is typically encoded in a Merkle tree, a data structure where all leaf nodes are values of one-way functions. Merkle trees are often found in cryptography. However, the Merkle tree has a particular downside: traversing the tree requires a large amount of computation [23].

A blockchain allows transactions without the need for a third party authority. The chain itself achieves a consensus if a new block is valid or not. This enables smart contracts, a technology that executes certain contract clauses when specified conditions are met. The contracts and their specifics are published on a blockchain and can be verified by other participants [24].

4.3.1.2 Using Blockchain to Create a Contract One possible way to create trust between the arbitrary PKIs in the authentication mesh is the use of a smart contract. The PKIs of the authentication mesh would be connected to a blockchain that spans over all participants in the mesh.

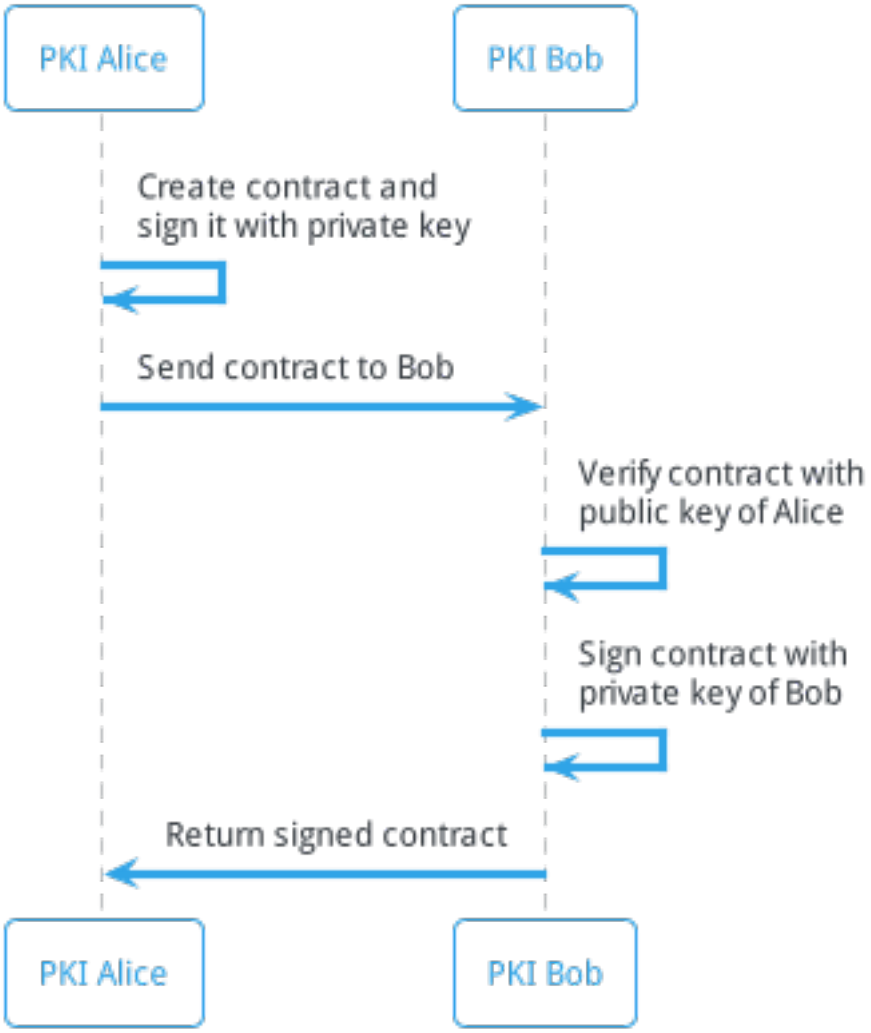

Figure 12: Blockchain Smart Contract between PKIs

Figure 12 shows the necessary steps to form trust between two PKIs in the authentication mesh. Since all operations are performed on a blockchain, the contract and the steps to create it are verified by other participants as well.

With the smart contract, both parties can exchange their public key material and generate a trust anchor between them without the need of a third party authority. As soon as the contract is voided by any of the parties, the trust anchor is revoked.

4.3.1.3 Using a Blockchain PKI to Create Certificates Another possibility to create trust between the distributed participants of the authentication mesh is the usage of

a distributed PKI (dPKI). The distributed PKI would act as a mediator between the different PKI that exist in each trust zone.

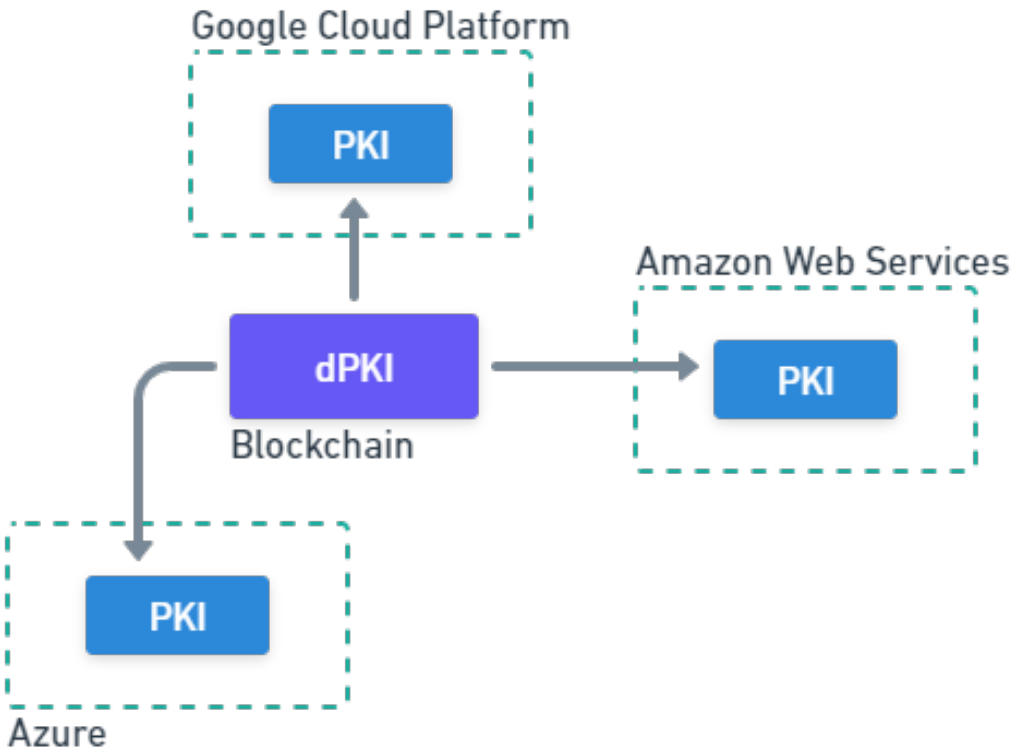

Figure 13: Using a Decentralized Public Key Infrastructure (dPKI) as root PKI to ensure that all participants are able to create trust between them.

With a dPKI deployed on a blockchain, as shown in Figure 13, each specialized PKI in a trust zone could request a certificate that acts as the root for the trust zone of that PKI. The PKI fulfills its role as key material provider for the specific zone and has knowledge about the other PKIs in the mesh through the blockchain. If two zones are to trust each other, a configuration on the blockchain defines that two parties must create trust. Since the specific PKIs already have the information about the other certificates, they can validate the public key material of services in other zones.

An example of such a distributed PKI for blockchain is “ETHERST”. However, deploying the PKI on the blockchain has the disadvantage of raising prices for the PKI. The participants need to pay the Ethereum gas fees to request and sign a certificate in ETHERST [25].

4.3.1.4 Security Issues with Blockchain

When considering the CIA triad in Section 2, only *integrity* and *availability* can be provided. No information that is published to the blockchain is confidential and can be read by all participants in the chain.

While the blockchain approach seems elegant, it also bears some security issues. A blockchain can be attacked by a "majority attack" where an attacker holds more than 51% of the computing power in the blockchain. If this happens, the next calculation for the Proof of Work algorithm can be found faster than the rest of the network is able to validate the calculation. Therefore, an attacker can decide which blocks are valid and which are not [26]. There exist other issues and attack vectors, but the majority attack would be the most threatening one for the Distributed Authentication Mesh.

4.3.2 Using a Master Node

A more centralized approach to form trust between participants is the usage of a master node.

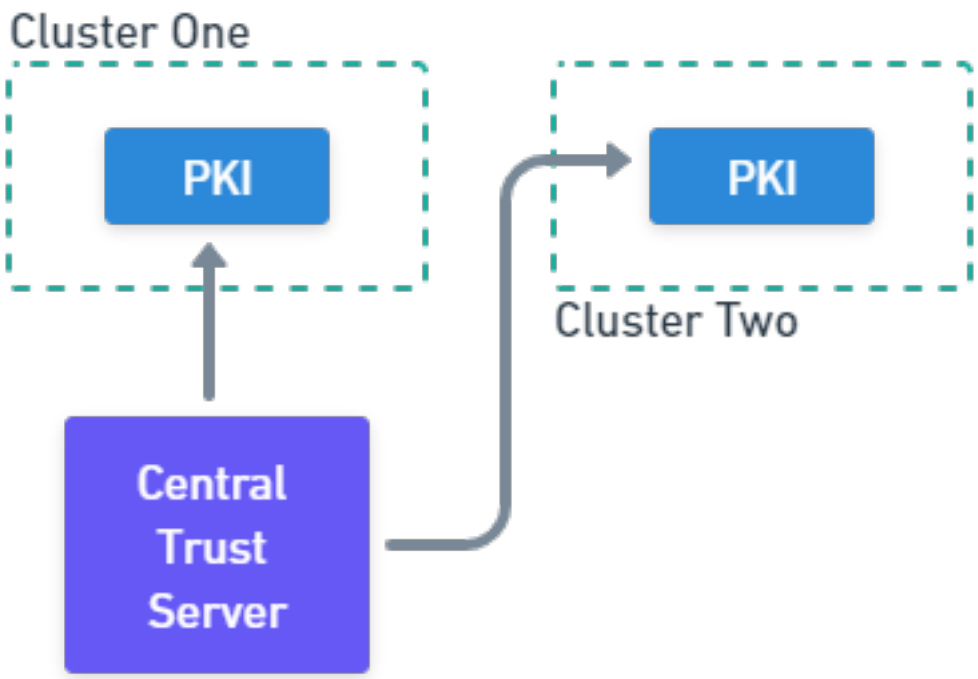

Figure 14: Centralized Trust Manager for Participants

Figure 14 shows the basic concept. While the trust zones remain decentralized, the

master node must be central to manage the trust between the PKIs. The master node creates contracts between the PKIs of the participants. This could happen via API calls or via configuration in a secure storage location. However, this creates a single point of failure since the master node must also validate the trust. Trust revocation is done via the master node as well. If the master node is the target of an attack, the whole trust in the mesh is threatened. The master node is the single point of failure for inter-zonal communication.

4.3.3 Distribute Contracts via Git

A third option to establish contracts between PKIs in the authentication mesh is the usage of a git repository. Git is a distributed version control system. It consists of a central repository server and a set of clients that clone the repository locally [27].

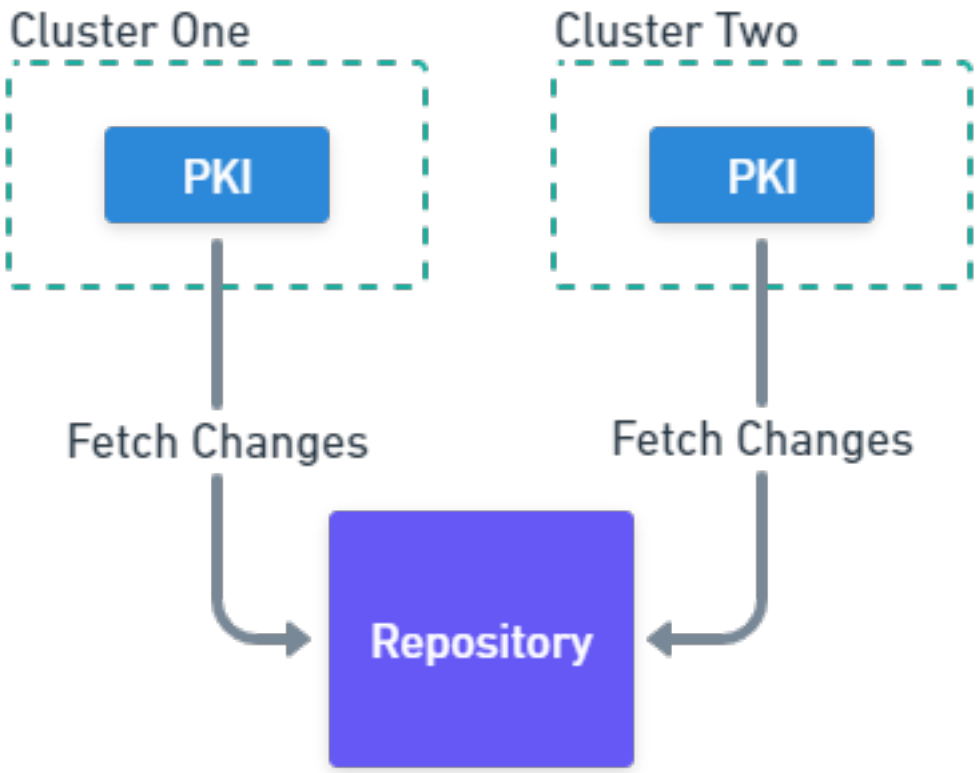

Figure 15: Use Git Repository for Trust Management

The basic principle is depicted in Figure 15. A central git repository acts as distribution

node for contracts between the parties and therefore between the trust zones. The contract is either created via some application or via manual creation by an administrator. The contract is then pushed into the central repository. All participants can periodically check for new or revoked contracts in the repository. A contract is only valid as long as the file is physically present in the repository. To revoke a contract, the file is deleted from the repository.

With a central repository, other security concerns arise. The repository is not crucial for the communication between participants, but it is relevant for the management of the contracts. While a denial of service attack may not impact the communication itself, it can disable the possibility to check for revoked contracts. Furthermore, the history of a git repository is not secure since the clients can hold a local clone.

4.4 Define the Contract

When considering all options in the previous section, using a combination between fetching contracts and having a master access point is a solid compromise. It does not require payment of blockchain gas fees nor the setup of a private blockchain. Furthermore, it does provide the possibility to create and revoke contracts while not being the single point of failure if the server does not respond for a certain time period. However, the central repository is not secure against denial of service attacks. Such attacks can disable the possibility to check for contract updates.

The most basic information that is required in the trust contract is the public certificate of the PKIs. The public certificate is the root certificate of the specific trust zone. When both parties have the public key of the other party, they are able to verify certificates of the other PKI and therefore are enabled to create mTLS (mutual TLS) connections. The usage of mTLS in the authentication mesh does ensure that only trusted connections are allowed and all other attempts to connect to a service are rejected. This further enables the authentication mesh to guarantee that only trusted participants can send the custom HTTP header that authenticates the user.

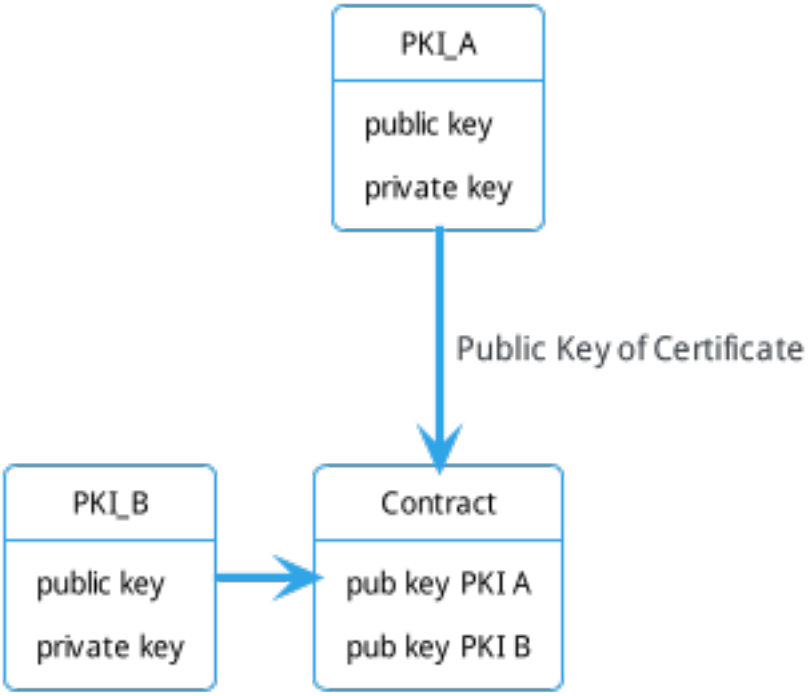

Figure 16: Trust Contract between PKIs

The contract between two parties is simple. As Figure 16 shows, the only parts required to form a contract is the public key of the respective partners. With the public key, either PKI can verify the other PKIs certificates and thus allow an mTLS connection. The contract can be extended in future work to enable other use-cases like rule based access control or other security features.

To enable serialization and to create a data scheme for the contracts, Protobuf[9] is used. Protobuf is a serialization format that defines the messages and calls in a `proto` file. The format is used by gRPC[10], a well-known RPC framework in microservice architecture. The `proto` files can be used to create client implementations and server stubs for programming languages.

```
message Participant {
    string name = 1;
    string public_key = 2;
    string hash = 3;
}
```

[9]https://developers.google.com/protocol-buffers
[10]Google Remote Procedure Call, https://grpc.io/

```
message Contract {
    repeated Participant participants = 1;
}
```

The `proto` definition above shows the structure of a contract. In principle, a contract is just a list of participants that trust each other. A participant may be involved in multiple contracts. All contracts that include the own participant, are fetched and installed into the local trust store. As soon as this is done, the Envoy proxy of the authentication mesh is able to connect to distant services with an mTLS connection.

4.5 Implementing the Contract Repository

The (open-source) implementation of the contract repository resides in the GitHub repository https://github.com/WirePact/k8s-contract-repository. The contract repository consists of two parts: "API" and "GUI". The separation of these parts is done to enable the usage of the API without the user interface. The contract provider only needs access to the API while an administrator could use the gRPC API or the graphical interface to manage the contracts.

4.5.1 Provide a High-Performance API for Contracts

The API is a gRPC based application that provides the means to fetch, create, and revoke contracts. The GUI is a web application that allows direct access to that API via web browser.

In contrast to a git based approach that is described in the previous sections, the local or Kubernetes storage provides a deterministic approach to store the contracts. Further, it improves the testability of the overall system. Using a git repository to store the contracts would not improve the security nor the distribution of the system. However, the basic concept of a git repository is used to distribute the contracts. The opposing part - the contract provider - fetches the contracts from the repository in a regular interval. The repository is not the single point of failure, but could be targeted with a denial of service attack.

The contracts do not contain any sensitive information. Therefore, the API does not need to encrypt them in any way. The contracts can be stored in two possible ways: "Local" and "Kubernetes". While the local storage repository just uses the local file system to store the serialized `proto` files, the "Kubernetes" storage adapter uses Kubernetes Secrets to store the contracts.

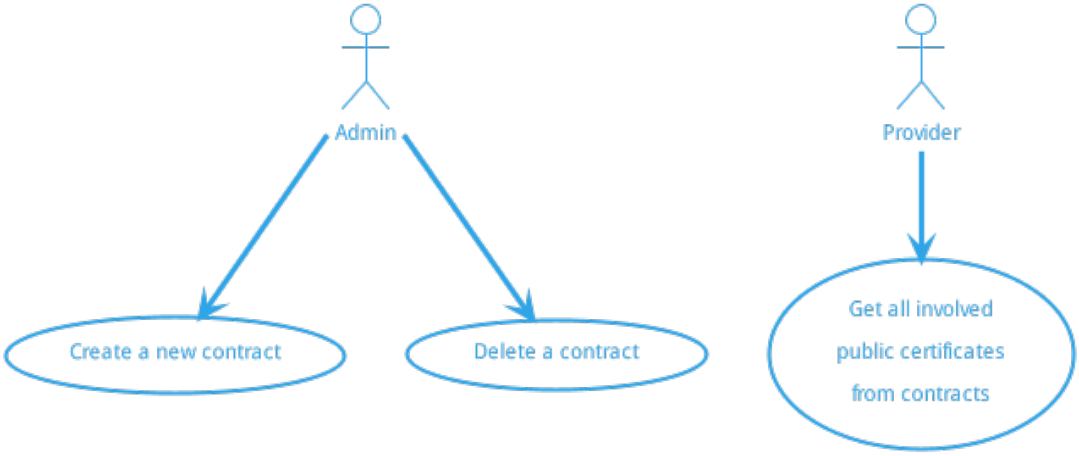

Figure 17: Use-cases for the Contract Repository

The use-cases shown in Figure 17 show the basic functionality of the contract repository. Admins use the GUI or the gRPC API to create, fetch, and revoke contracts in the system. Providers then use the gRPC API to fetch a list of all involved public certificates. This allows the contract provider to create a certificate chain that contains all involved parties and therefore allows mTLS connections to corresponding services.

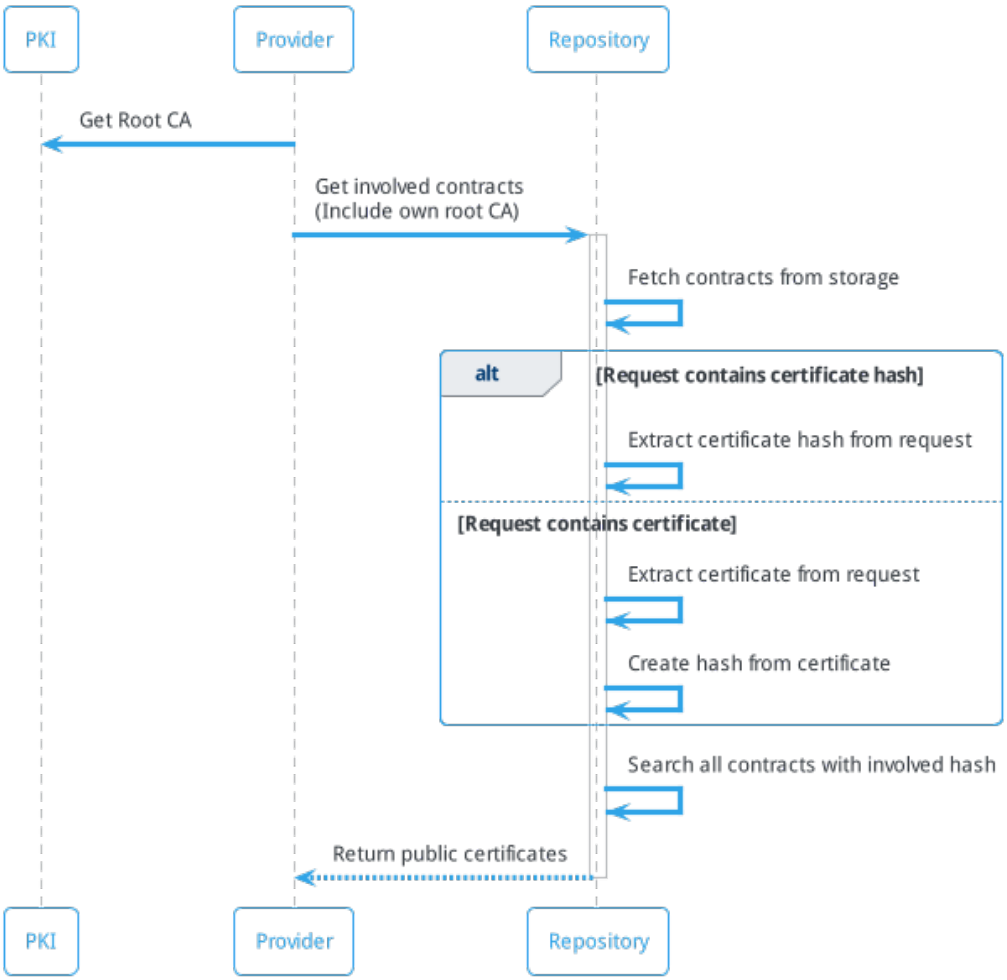

Figure 18: Provider fetching relevant contracts from the repository

The application sequence in Figure 18 depicts the process when a provider fetches the relevant list of contracts for itself. The provider calls the repository with its own public certificate (which it fetches from its own PKI). The repository then returns a list of all contracts that the provider is part of.

4.5.2 Administrate Contracts via Graphical Web Interface

The GUI application is based on the "Lit"[11] framework, which uses native web components to create applications instead of an engine like "React" and "Angular". Web components

[11] https://lit.dev/

are a mix between different technologies to create reusable custom HTML elements. They consist of three main technologies ("Custom HTML Elements", "Shadow DOM", and "HTML Templates") to create reusable elements with encapsulated functionality [28].

```
import { html, css, LitElement } from 'lit';
import { customElement, property } from 'lit/decorators.js';

@customElement('demo-element')
export class DemoElement extends LitElement {
  static styles = css`
    p {
      color: pink;
    }
  `;

  @property()
  name = 'World';

  render() {
    return html`<p>Hello ${this.name}!</p>`;
  }
}
```

The code above creates a custom "demo-element" that just prints "Hello World!" in pink. Note that the CSS style is not interfering with any other styles. The CSS block is encapsulated in this particular component only. To use the component above, one needs to include the "demo-element" in their HTML code.

```
<div>
  <demo-element></demo-element>
</div>
```

The HTML above will render the demo element component inside the `<div>` and print "Hello World!" in pink. If multiple of these components are rendered, each has its own root DOM such that there is no interference between them.

The GUI application of the contract repository will allow administrators to create and delete contracts in the repository. The GUI directly interacts with the repository via gRPC-web calls. In contrast to gRPC, gRPC-web is a protocol that allows the usage of gRPC in web applications. It allows HTTP/1.1 and HTTP/2 calls and requires the API to understand gRPC-web or any form of translation layer between the two protocols.

4.6 Implementing a Contract Provider

The contract provider is an application that fetches the contracts from the repository in a defined interval. The implementation can be found on the GitHub repository https://github.com/WirePact/k8s-contract-provider.

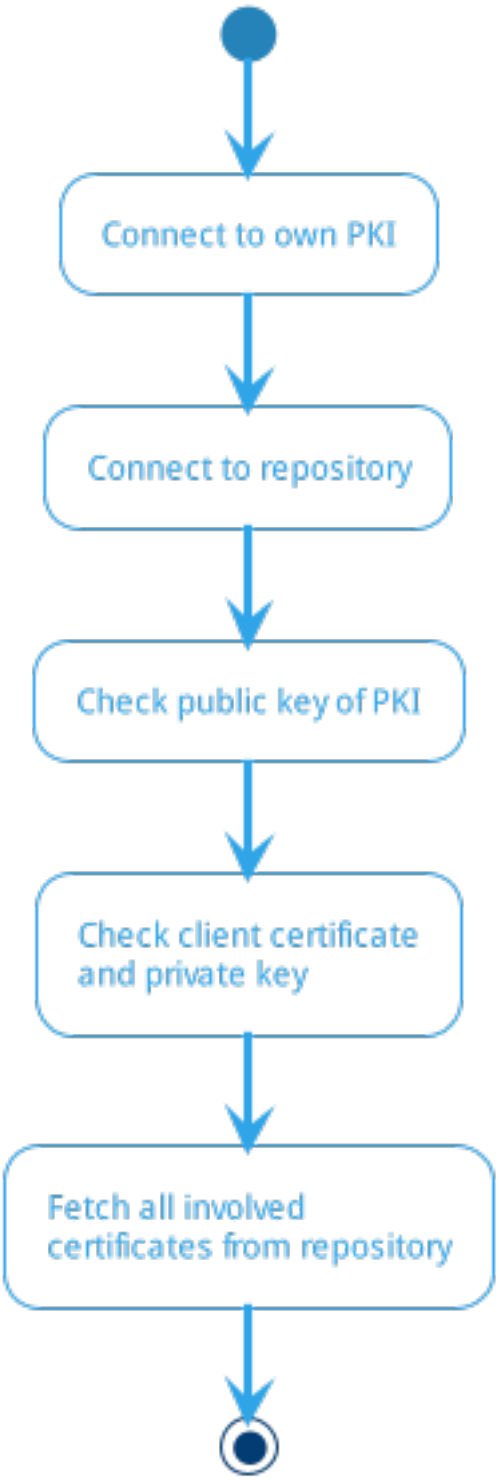

Figure 19: Activity of the provider during each interval

During each interval, the provider executes the steps in Figure 19:

1. Connect to its own PKI.
2. Connect to the contract repository.
3. Check if the public key of the PKI is stored, if not, download and store it.
4. Check if a client certificate and key are stored, if not, create a key and fetch a certificate from the PKI.
5. Fetch all public certificates that the "own PKI" is involved it and store the certificates.

The following code blocks describe the actions that the provider takes to achieve the steps above.

```
debug!("Check PKI public certificate.");
if !storage.has_ca().await {
    info!("Fetching PKI public certificate.");
    let response = pki.get_ca(Request::new(())).await?.into_inner();
    storage.store_ca(&response.certificate).await?;
}
```

The first step after connecting to the PKI and the contract repository is to check if the configured storage location contains the public certificate of the "own" PKI. If not, the provider fetches the public certificate from the PKI and stores it in the storage adapter.

```
debug!("Check private certificate.");
if !storage.has_certificate().await {
    info!("Sign private certificate.");
    let (key, csr) = create_csr(&config.common_name)?;
    let response = pki
        .sign_csr(Request::new(grpc::pki::SignCsrRequest {
            csr: csr.to_pem()?,
        }))
        .await?
        .into_inner();
    storage
        .store_certificate(
            &response.certificate,
            &key.private_key_to_pem_pkcs8()?,
        )
        .await?;
}
```

Next, the provider validates if a client certificate and key are present in the storage adapter. This client certificate is required to enable the Envoy proxy to present it for the mTLS connection to the distant serivce. If no certificate and/or key is found, the

provider creates a new key and a certificate signing request (CSR) and sends it to the PKI. The PKI then signs the CSR and returns the signed certificate. The provider now stores the certificate and the key in the storage adapter.

```
debug!("Fetch certificate chain.");
let (ca, ca_hash) = storage.get_ca().await?;
let response = repo
    .get_certificates(Request::new(
        grpc::contracts::GetCertificatesRequest {
            participant_identifier: Some(
                ParticipantIdentifier::Hash(ca_hash)
            ),
        }
    ))
    .await?
    .into_inner();
let mut certificates = response.certificates;
certificates.push(ca);
storage.store_chain(&certificates).await?;
info!("Stored {} certificates in chain.", certificates.len());
```

The last step is to fetch all certificates that are involved in the contracts that the provider is part of. The provider loads the public certificate of the "own" PKI and uses the hash of the certificate to fetch all participants that share a contract with its own PKI. The provider then attaches its own PKI root CA into the chain (since the API only returns "other" certificates) and stores the chain in the storage adapter.

Like other applications in this project and the Distributed Authentication Mesh, the provider is able to store the certificates in a local or Kubernetes storage adapter. The main goal of the provider is to fetch all public keys of participating PKIs to enable mutual TLS (mTLS) connections between participants.

Since there are multiple possible ways to inject additional trusted root certificates (all participant PKIs), the provider does only store the certificate in the defined storage adapter. In Kubernetes and its ingress controllers, the TLS context must be configured to use the certificate, the key, and the trusted root certificates. The NGINX ingress controller must know where the client certificate resides to connect to an internal service.

4.7 Create Secure Communication between Services

With the Distributed Authentication Mesh and the additional extensions of this project, we are now able to create fully trusted communication between distant services. Even if the applications are not running in the same trust context. The Distributed Authentication Mesh provides the means to create a signed identity that can be used to authenticate a user

[1]. The common identity allows participating systems to restore required authorization information for the targeted service [2].

The contract repository and provider now allow the PKIs to form a trust contract with each other. This in turn allows services to establish mTLS connections with each other. When participants of the mesh communicate with other services in distant trust contexts, mTLS ensures that only allowed connections can be created. This mitigates the risk of external services forging an identity and connect to internal services. The secured connection proofs that the PKIs are trusted and therefore no further encryption for the common identity is required. The mTLS connection cannot be successfully created if the service (respectively its PKI) is not involved in a contract with the destination.

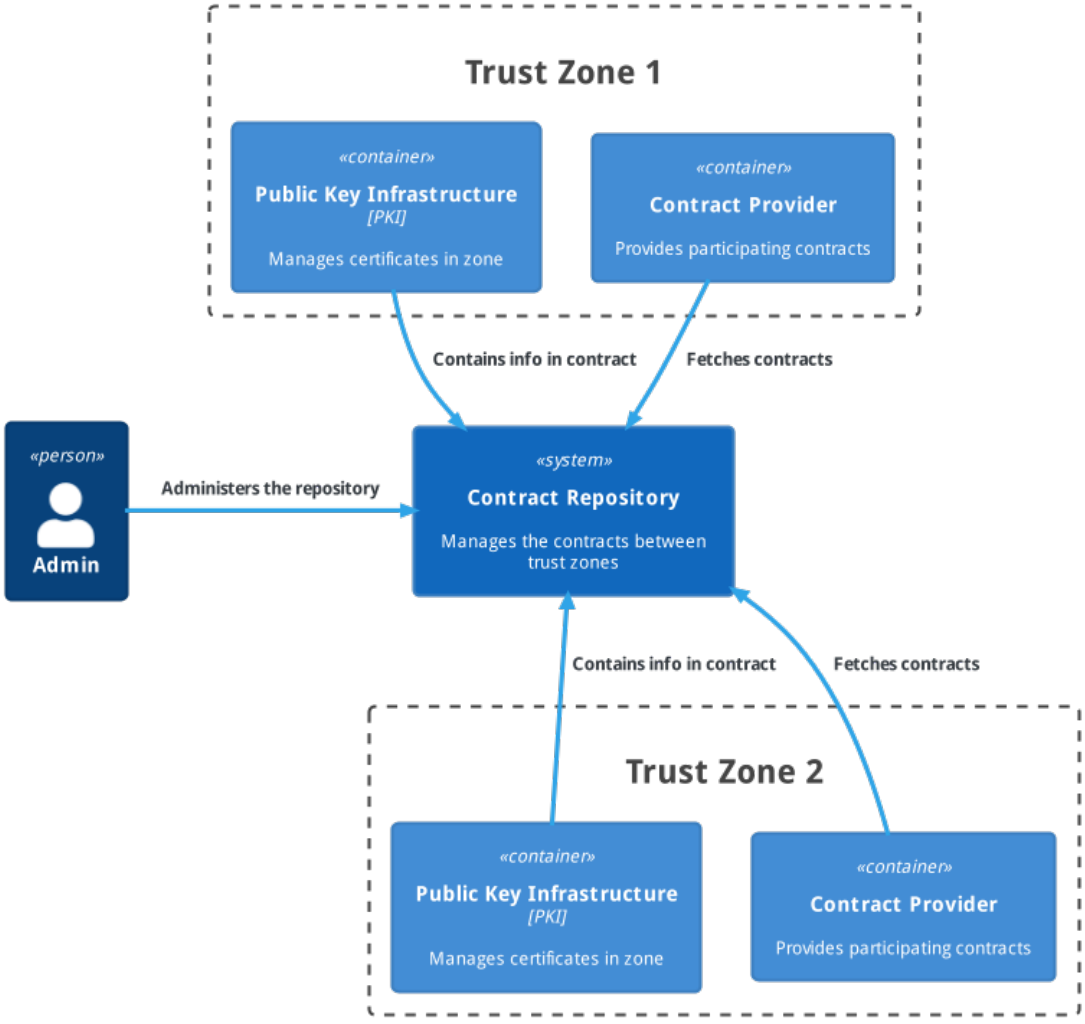

Figure 20: The Contract Repository and the Trust Zones

Figure 20 shows how the parts interact with the contract repository. There are two different trust zones, each of which contains its own "main" PKI. The PKI generate a CA certificate root and create client certificates for the services within the same trust zone. An admin can create a trust contract between the two trust zones and stores the contract in the repository. Contract providers (for each service) can then fetch the contracts and provide a client certificate and a certificate chain to validate incoming client certificates.

4.8 A Trusted Distributed Authentication Mesh

One issue with the Distributed Authentication Mesh is that the identity of a user is sent to a specific target service. The destination then translates this identity into valid authentication credentials [1]. This target service has no means to verify that the sender is actually part of the mesh itself [2]. Inside the same trust zone, the service can trust the sender if it is not publicly exposed. But, the use-case of the mesh includes communication between different trust zones. Therefore, the service must be able to verify that the sender is part of the mesh. With the mentioned contracts and the contract repository, it is possible for all participants to fetch a list of contracts. The contracts include the public certificates of all participating PKIs. Thus, it is possible for an application to call an API in a distant trust zone and verify that the sender is part of the mesh.

To show and verify the statement, a demo application setup in Docker is provided in the GitHub repository "https://github.com/WirePact/docker-demo". This demo proofs that it is possible to create a connection between two applications via mTLS connection.

The Docker demo consists of various containers that are required for the mesh. To verify the setup and the system itself, this section provides a step by step analysis of the demo and the functionality of the mesh in conjunction with the contract repository.

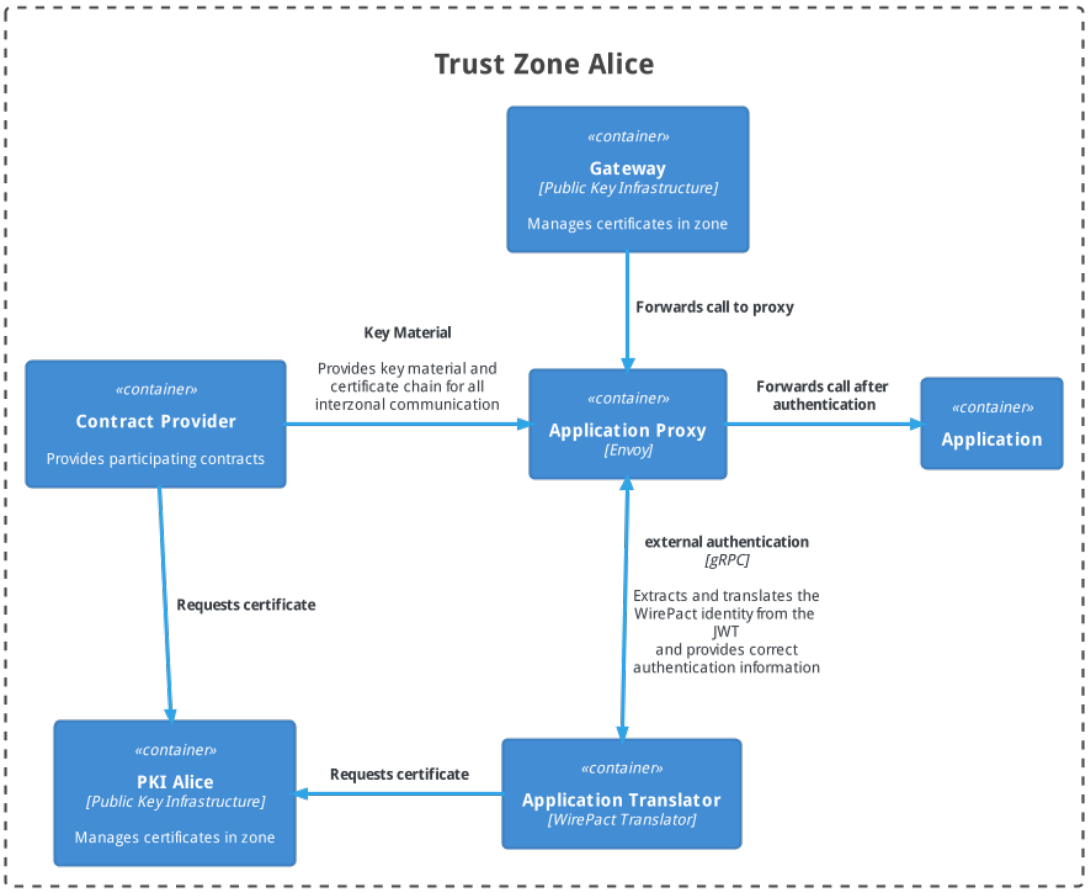

Figure 21: Trust Zone Alice

Figure 21 shows the setup for the first trust zone, “Trust Zone Alice”. It consists of a PKI, a contract provider, the application, an application proxy and the translator for WirePact. The PKI creates its own root certificate authority (CA) and creates a client certificate for the contract provider and the translator. The translator is responsible for the extraction and translation of the WirePact common identity [2]. The proxy manages all incoming and outgoing communication of the application itself. To enable general access to the application, a public gateway allows incoming communication and passes it to the application proxy.

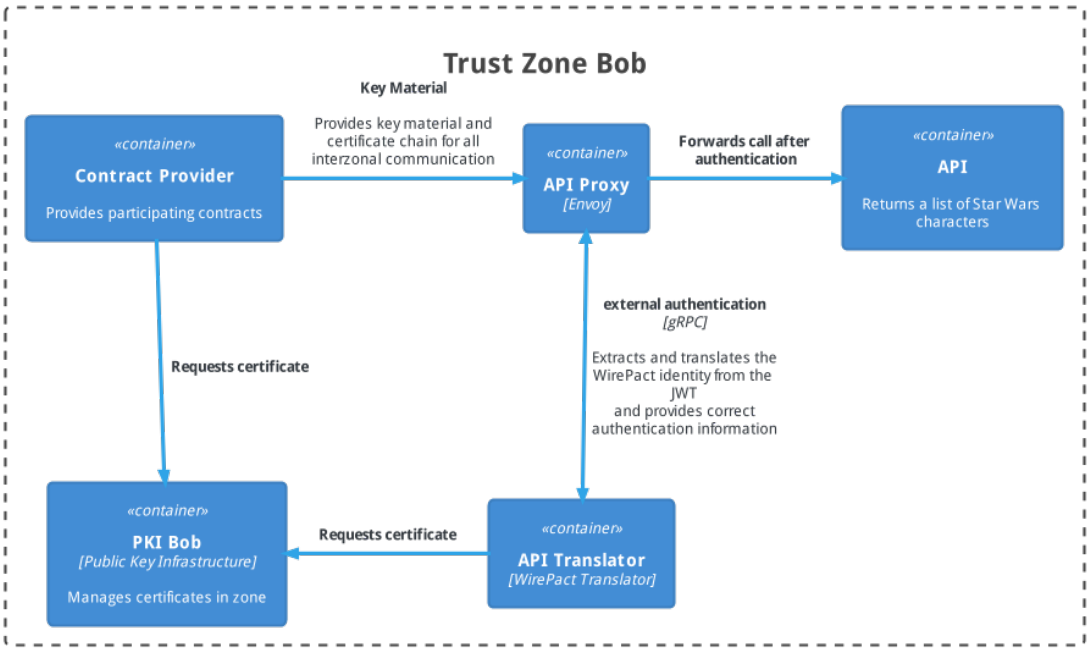

Figure 22: Trust Zone Bob

The second trust zone, depicted in Figure 22, is similar. It contains the same elements except for a public gateway since the demo system resides in Docker. A real world example would include another gateway that limits the access to other containers in the system.

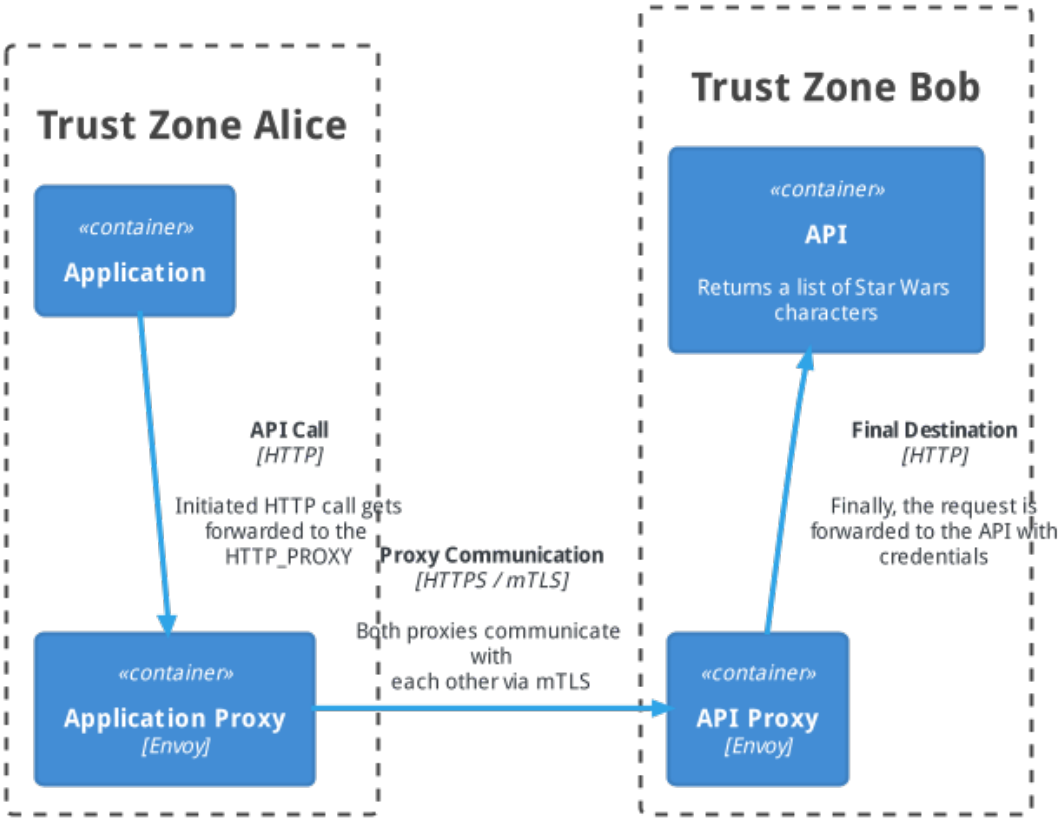

Figure 23: Communication between Trust Zones

Without a contract, communication as shown in Figure 23 is not possible. The HTTPS / mTLS connection between the two proxies cannot be established since they have totally different root CAs. To enable communication between the parties, both proxies must now all public certificates of the involved parties to allow verification of the certificates. When the contract is created, the public certificates of both PKIs are inserted and then stored in the contract repository. Both contract providers will fetch the contract and deliver the full certificate chain to their respective proxies. The proxies can now verify the certificates and establish a connection.

```
TCP        66 47518 → 9000 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2097607191 TSecr=1827478417
TLSv1.2   196 Client Hello
TCP        66 9000 → 47518 [ACK] Seq=1 Ack=131 Win=65152 Len=0 TSval=1827478417 TSecr=2097607191
TLSv1.2  2213 Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
TCP        66 47518 → 9000 [ACK] Seq=131 Ack=2148 Win=63872 Len=0 TSval=2097607192 TSecr=1827478418
TLSv1.2  2115 Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message
TCP        66 9000 → 47518 [ACK] Seq=2148 Ack=2180 Win=63872 Len=0 TSval=1827478419 TSecr=2097607193
TLSv1.2  2020 New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
TCP        66 47518 → 9000 [ACK] Seq=2180 Ack=4102 Win=64000 Len=0 TSval=2097607194 TSecr=1827478420
TLSv1.2  4481 Application Data
TCP        66 9000 → 47518 [ACK] Seq=4102 Ack=6595 Win=62720 Len=0 TSval=1827478420 TSecr=2097607194
TLSv1.2  1337 Application Data
TCP        66 47518 → 9000 [ACK] Seq=6595 Ack=5373 Win=64128 Len=0 TSval=2097607586 TSecr=1827478812
```

Figure 24: mTLS Connection between Proxies

To proof that the connection is secured via mTLS, the network traffic of the demo Docker setup was recorded[12]. Figure 24 shows the TLS handshake between the two proxies. All other communication is HTTP, while the communication between the proxies is HTTPS. We can see that the server does present its own certificate accompanied by the certificate request for the client. The client in turn does present its own certificate and then, the connection is established.

[12]With “termshark”, a terminal only alternative to Wireshark (https://github.com/gcla/termshark)

5 Conclusions and Outlook

In this thesis, we presented a solution to truly distribute the authentication mesh over distant clusters and trust zones. The Distributed Authentication Mesh in conjunction with the common identity was not able to run safely across trust zones [1], [2]. Inside the same trust zone, the mesh did provide its functionality, but as soon as the communication spans multiple clusters with their own gateways, the trust between the participants of the mesh could not be guaranteed. There was no mechanism in place to verify the sender of a common identity. The goals of this work were to analyze the situation of distributed trust systems and provide a solution for the Distributed Authentication Mesh.

Section 1 introduces the reader into the topic and gives references and introductions to past work. The section also briefly describes the general issue with shared trust in the Distributed Authentication Mesh.

Section 2 defines the scope of this thesis and presents prerequisite knowledge to the reader. The section gives an introduction into Kubernetes, the container orchestration software, since it is used to show the use-case of the Distributed Authentication Mesh. Furthermore, several security related topics are described to allow the reader to understand the general problems at hand. Section 2 also explains two basic authentication and authorization mechanisms: “HTTP Basic” and “OpenID Connect”. These mechanisms are primarily used to present the use-case of the Distributed Authentication Mesh.

Section 3 then describes the current state of the Distributed Authentication Mesh and its shortcomings. It shows how the mesh uses the common identity from [2] to encode authorization information into a JSON Web Token (JWT). The section then gives a general hypothesis on how the communication between distant trust zones can be achieved.

Section 4 describes the main part of this thesis. The section introduces the reader into the used demo applications, technologies, programming languages and concepts of the thesis. A short introduction into the Rust programming language explains why the language was chosen for the implementation. Then, three different possibilities to create a contract between two parties are explained and analyzed. After the explanation of the considered methods, the contract itself is designed and defined. The section then shows how the contract repository and the contract provider - two essential elements of this work - are implemented. Finally, Section 4 validates and proofs that the trusted connection between multiple PKI participants works as expected.

This thesis analyzed the issue and gives a detailed solution for the problem. The solution is based on the idea of a trust contract between the public key infrastructures (PKIs) of the mesh. Each PKI creates its own root certificate authority (CA) which is used by the services inside the same trust zone to communicate with each other. To allow communication between distant trust zones, the services must know the public certificates of the distant PKI to validate the provided client certificate. When such a contract is created, the contract provider fetches the involved public certificates and provides

the certificate chain as a file. With that, the application proxies are able to require a client certificate and can validate it against the provided certificate chain. This allows the Distributed Authentication Mesh to span multiple trust zones and to be truly distributed.

All created implementations and software is available as open-source under the Apache 2.0 license. The source code can be found on GitHub under the organization "WirePact" (https://github.com/WirePact). To run a working demo setup of the full Distributed Authentication Mesh, the reader can follow the instructions in the README of the "docker-demo" repository (https://github.com/WirePact/docker-demo).

Future work can include the purposed work of past work like the addition of a rule engine into the concept of the Distributed Authentication Mesh. These rules could be incorporated into the contracts that are purposed in this thesis. Such a rule engine could, as an example, allow time based access to services inside a trust zone and block the communication when the rules do not apply. Also, with this project, the Distributed Authentication Mesh is distributed and can communicate accross trust zone boundaries. However, the contract repository itself is not yet truly distributed. Future work can include the distribution of the contract repository with a distribution of the contract database itself.

Bibliography

[1] C. Bühler, "Distributed Authentication Mesh - A Concept for Declarative Ad Hoc Conversion of Credentials," University of Applied Science of Eastern Switzerland (OST), Aug. 2021. Available: https://buehler.github.io/mse-project-thesis-1/report.pdf

[2] C. Bühler, "Common Identities in a Distributed Authentication Mesh - Definition and Implementation of a Common Identity for Secure Transport," University of Applied Science of Eastern Switzerland (OST), Feb. 2022. Available: https://buehler.github.io/mse-project-thesis-2/report.pdf

[3] B. Burns, J. Beda, and K. Hightower, *Kubernetes*, Second Edition. Dpunkt Heidelberg, Germany, 2018.

[4] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site reliability engineering: How google runs production systems*. " O'Reilly Media, Inc.", 2016.

[5] J. Dobies and J. Wood, *Kubernetes operators: Automating the container orchestration platform*. O'Reilly Media, 2020.

[6] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," Jun. 2016. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns

[7] S. Samonas and D. Coss, "The CIA strikes back: Redefining confidentiality, integrity and availability in security." *Journal of Information System Security*, vol. 10, 2014.

[8] A. Mallik, "Man-in-the-middle-attack: Understanding in simple words," *Cyberspace: Jurnal Pendidikan Teknologi Informasi*, vol. 2, no. 2, pp. 109–134, 2019, doi: http://dx.doi.org/10.22373/cj.v2i2.3453.

[9] I. Ahmed, T. Nahar, S. S. Urmi, and K. A. Taher, "Protection of sensitive data in zero trust model," 2020. doi: 10.1145/3377049.3377114.

[10] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero trust architecture," National Institute of Standards; Technology, 2019.

[11] J. Reschke, "The 'Basic' HTTP authentication scheme," Internet Engineering Task Force IETF, RFC, Sep. 2015. doi: 10.17487/RFC7617.

[12] D. Hardt *et al.*, "The OAuth 2.0 authorization framework," Internet Engineering Task Force IETF, RFC, Oct. 2012. doi: 10.17487/RFC6749.

[13] N. Sakimura, J. Bradley, M. Jones, B. De Medeiros, and C. Mortimore, "Openid connect core 1.0," The OpenID Foundation OIDF, Spec, 2014. Available: https://openid.net/specs/openid-connect-core-1_0.html

[14] P. Siriwardena, "Mutual authentication with TLS," in *Advanced API security: Securing APIs with OAuth 2.0, OpenID connect, JWS, and JWE*, Berkeley, CA: Apress, 2014, pp. 47–58. doi: 10.1007/978-1-4302-6817-8_4.

[15] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2," Internet Engineering Task Force IETF, RFC, Aug. 2008. Available: https://tools.ietf.org/html/rfc5246

[16] J. Richer, "OAuth 2.0 Token Introspection," Internet Engineering Task Force IETF, RFC, Oct. 2015. doi: 10.17487/RFC7662.

[17] M. Jones, A. Nadalin, B. Campbell, J. Bradley, and C. Mortimore, "OAuth 2.0 Token Exchange," Internet Engineering Task Force IETF, RFC, Jan. 2020. doi: 10.17487/RFC8693.

[18] S. Klabnik and C. Nichols, *The rust programming language (covers rust 2018)*. No Starch Press, 2019.

[19] N. Ivanov, "Is rust c++-fast? Benchmarking system languages on everyday routines," 2022, doi: 10.48550/ARXIV.2209.09127.

[20] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017, doi: 10.1145/3158154.

[21] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in rust," *Communications of the ACM*, vol. 64, no. 4, pp. 144–152, 2021.

[22] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, "Blockchain," *Business & Information Systems Engineering*, vol. 59, no. 3, pp. 183–187, 2017.

[23] M. Jakobsson, T. Leighton, S. Micali, and M. Szydlo, "Fractal merkle tree representation and traversal," in *Topics in cryptology — CT-RSA 2003*, 2003, pp. 314–326.

[24] Z. Zheng *et al.*, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020, doi: https://doi.org/10.1016/j.future.2019.12.019.

[25] C.-G. Koa, S.-H. Heng, and J.-J. Chin, "ETHERST: Ethereum-based public key infrastructure identity management with a reward-and-punishment mechanism," *Symmetry*, vol. 13, no. 9, 2021, doi: 10.3390/sym13091640.

[26] I.-C. Lin and T.-C. Liao, "A survey of blockchain security issues and challenges." *International Journal of Network Security*, vol. 19, no. 5, pp. 653–659, 2017, doi: 10.6633/IJNS.

[27] D. Spinellis, "Git," *IEEE Software*, vol. 29, no. 3, pp. 100–101, 2012, doi: 10.1109/MS.2012.61.

[28] MDN Contributors, "Web Components." Mozilla Foundation, Aug. 2022. Available: https://developer.mozilla.org/en-US/docs/Web/Web_Components