TODO*

Definition and Implementation of a Common Identity for Secure Transport

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 “TODO” 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, April 25, 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 only use one trust zone[1]. While still allowing “zero trust”[2], 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].

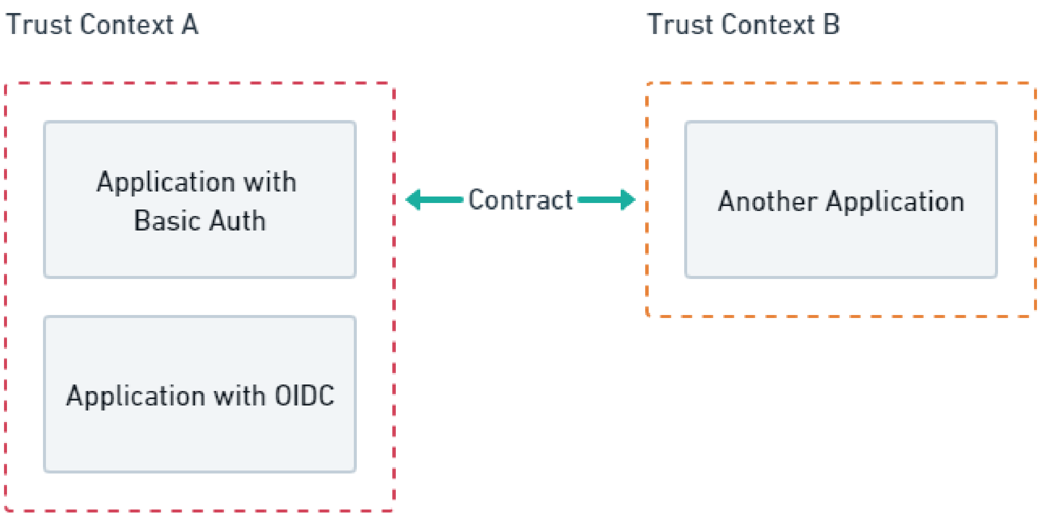

Figure 1: Multiple trust zones that share a contract between them. The contract enables the authentication mesh to verify callers from other zones.

To achieve true distribution, a contract as shown in Figure 1 must exist. The contact defines how multiple parties can trust each other. This project shall define and implement

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

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 authentication mesh is provided. The implementation runs on Kubernetes[3] and is automated by a Kubernetes Operator.

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 conclusions section then gives an overview of the results and provides an outlook into future work.

[3] https://kubernetes.io

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 technologies 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 Kubernetes. 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 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.

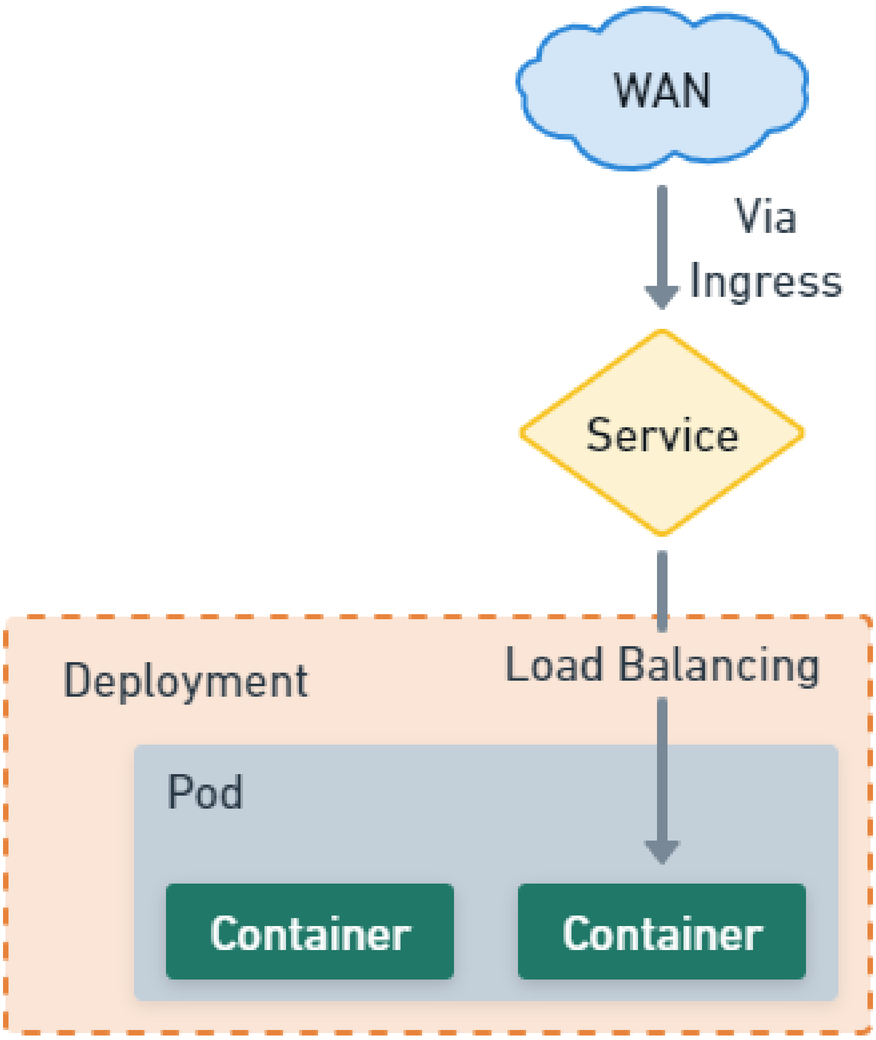

Figure 2: Basic Buildingblocks in Kubernetes

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”[4], a container runtime for various platforms.

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

[4]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"[5] 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"[6] 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.

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

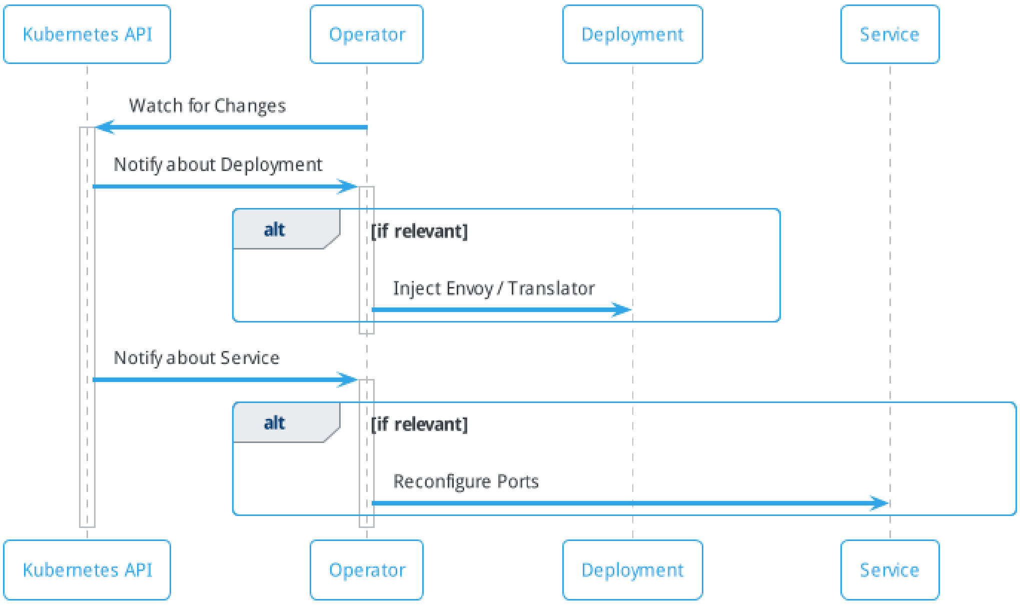

Figure 3: Basic Buildingblocks in Kubernetes

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[7] 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]

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

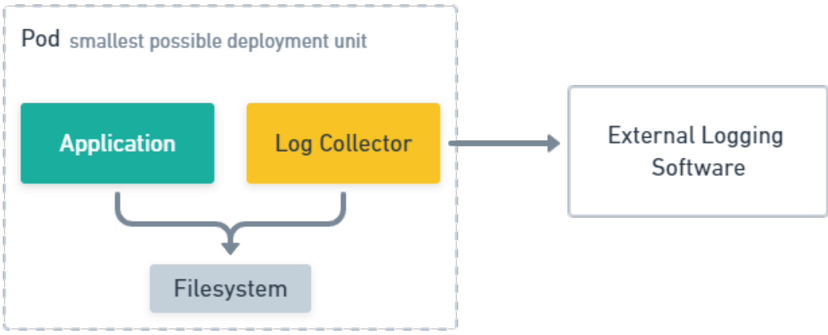

Figure 4: An example of a Sidecar

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[8].

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 is one of the main focus 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

[8] 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. An attacker that 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 attack 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.

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].

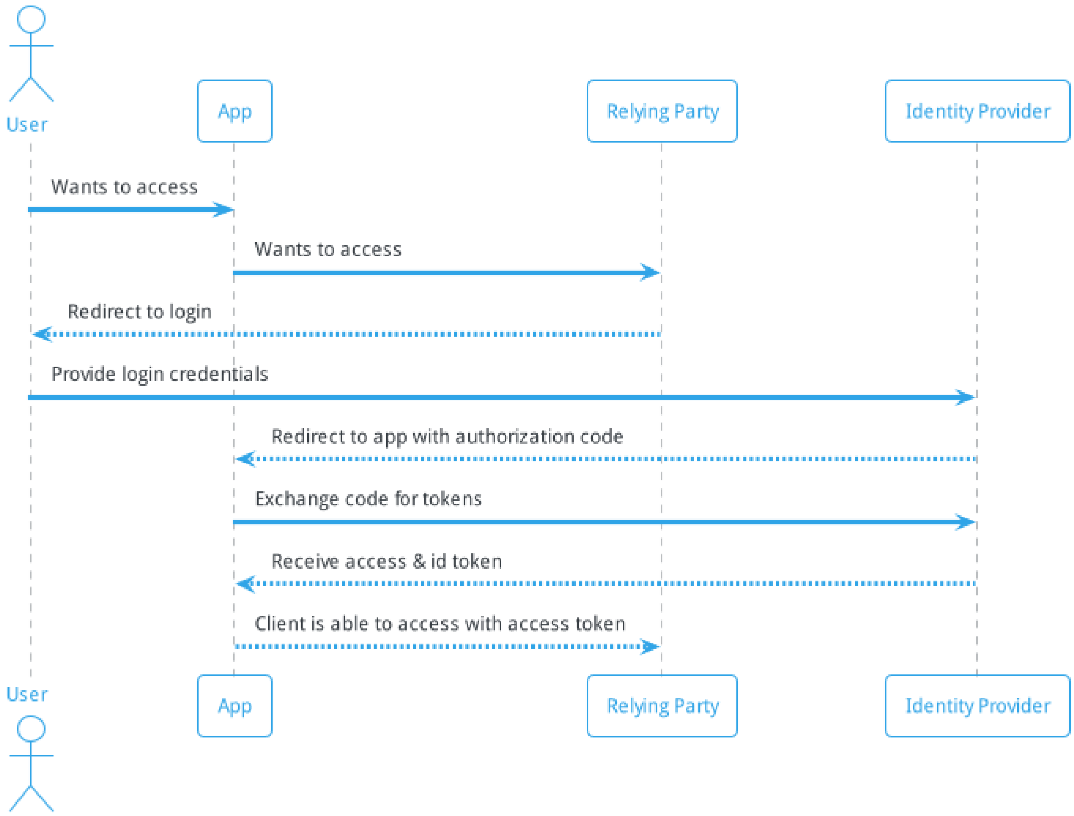

Figure 5: OIDC code authorization flow [13]. Only contains the credential flow, without the explicit OAuth part. OAuth handles the authorization whereas OIDC handles the authentication.

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].

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 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 JSON Web Token (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 (public key infrastructure) 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.

3.2 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 they “speak” with the correct entity and thus can verify if a trust anchor between the two exists.

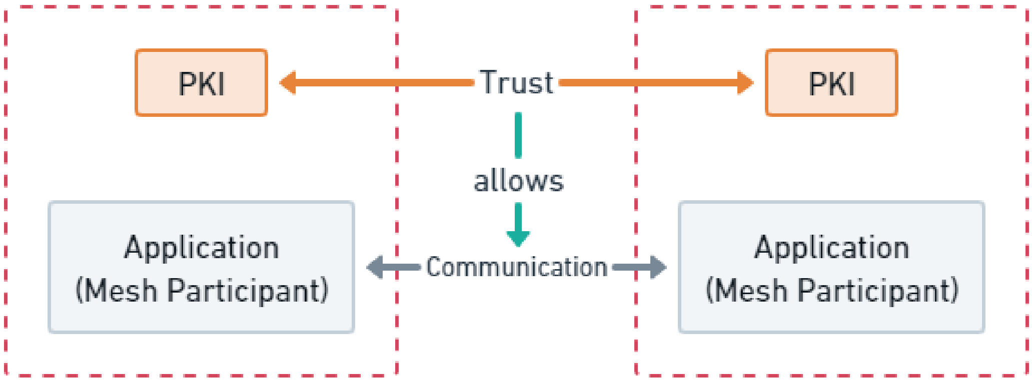

Figure 6: Creating Trust with a Contract

Regarding Figure 6, a contract between the two trust zones creates the trust anchor between the zones. This trust further allows an mTLS connection 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 used demo applications, the programming language Rust, and several security topics that are relevant for the implementation of the authentication mesh. Furthermore, the implementation of the shared trust context is described.

4.1 Demo Applications

To demonstrate and test the implementation of the trust context, 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[9]. 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 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_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. The application runs in an ASP.NET context. Thus, it is possible to respect the `HTTP_PROXY` variable and route traffic through a specific proxy.

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 [15]. The API needs an issuer, a client ID, and a client secret to validate the given tokens. The configuration of the C# application is done as follows:

[9] https://go.dev/

```
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,
        };
    });
```

To complement the OIDC API, an **oidc_app** provides the means to access an OIDC (OAuth) 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_PROXY` is respected.

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 [16].

4.2 The Rust Programming Language

To achieve the goals of this work, the programming language “Rust” provides the necessary features to implement the authentication mesh. Rust itself is a multi-paradigm language that supports object-oriented features as well as functional components. Rust allows low-level memory management without the need for garbage collection. To achieve this, Rust uses a special type checking mechanism that allows the compiler to calculate the lifetime of references [17].

With the calculation of ownership and the transfer of ownership, Rust ensures that no “null-pointer” can exist, and no object can be manipulated without specifically taking ownership. A formal type system, named “RustBelt”, provides insight into the type checking mechanisms of Rust. It further shows that the programming language can guarantee memory safety [18].

To demonstrate the advantages of Rust, consider the following code examples taken from the article “Safe Systems Programming in Rust” [19]:

```
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" [19].

```
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." [19].

The safety of the Rust programming language and the C++-like performance are the primary reasons for the choice of the language.

4.3 Sign and Distribute Contracts between Participants

4.3.1 Using a Block Chain

4.3.1.1 Introduction

4.3.2 Using a Master Key

4.3.3 Distribute Contracts via Git

4.4 Define the Contract

5 Conclusions and Outlook

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

[16] 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.

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

[18] 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.

[19] 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.