Common Identities in a Distributed Authentication Mesh*

Definition and Implementation of a Common Identity for Secure Transport

Christoph Bühler

Autumn Semester 2021
University of Applied Science of Eastern Switzerland (OST)

The "Distributed Authentication Mesh" in [1] is a concept to dynamically convert authentication information (such as access tokens from OpenID Connect) to other authentication schemes (like HTTP Basic). In contrast to "Security Assertion Markup Language" (SAML), the concept does not require all participants to share the same authentication scheme. It eliminates the requirement to introduce code changes into existing applications such that they can support other authentication schemes.

A central part of the mesh is the "common language format". This format is eminently important to the mesh because it delivers the users' identity to other participants. While the previous project included the concept of the mesh and implemented a Proof of Concept for the modification of HTTP headers, it did not provide a definition nor implementation for the common language format.

This project targets the topic of the common language and analyzes several possibilities for such a format. The project also defines the objects that must be transmitted between mesh participants. The concept of the mesh is extended with a "Rule Engine" that improves the security and versatility of the mesh. Additionally, this project implements the "Distributed Authentication Mesh" as open-source software such that it can be operated on Kubernetes. The conclusion provides further information about the project and possible topics of follow-up work.

*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

Declaration of Authorship **5**

1 Introduction **6**

2 Definitions and Clarification of the Scope **8**
2.1 Scope of the Project . . . 8
2.2 Kubernetes and its Patterns . . . 9
2.2.1 Terminology of Kubernetes . . . 9
2.2.2 Kubernetes, the Orchestrator of Software . . . 9
2.2.3 An Operator, the Reliability Engineer . . . 10
2.2.4 A Sidecar, the Extension to a Pod . . . 12
2.3 Securing Communication . . . 12
2.3.1 HTTP Basic Authentication . . . 13
2.3.2 OpenID Connect . . . 13
2.3.3 Trust Zones and Zero Trust . . . 14

3 State of the Authentication Mesh **16**

4 Implementing a Common Language and a Mesh Operator **18**
4.1 Goals and Non-Goals of the Project . . . 18
4.2 A Way to Communicate with Integrity . . . 18
4.2.1 YAML, XML, JSON, and Others . . . 19
4.2.2 X509 Certificates . . . 19
4.2.3 JSON Web Tokens . . . 20
4.2.4 Using JWT in the Authentication Mesh . . . 20
4.3 A Public Key Infrastructure as Trust Anchor . . . 21
4.3.1 “Gin”, a Go HTTP Framework . . . 22
4.3.2 Prepare the CA . . . 22
4.3.3 Deliver the CA . . . 24
4.3.4 Process Certificate Signing Requests (CSR) . . . 25
4.3.5 Authentication and Authorization against the PKI . . . 27
4.4 Provide a Translator Base . . . 27
4.4.1 Define the Common Identity . . . 27
4.4.2 Startup a Translator . . . 28
4.4.3 Provide Endpoints for Interception . . . 30
4.4.4 Encode the JWT . . . 32
4.4.5 Decode the JWT . . . 32
4.5 Implementing an HTTP Basic Translator with a Secure Common Identity 33
4.5.1 Validate and Encode Outgoing Credentials . . . 34
4.5.2 Validate and Decode an Incoming Identity . . . 36
4.5.3 Contrast to an OIDC Translator . . . 39

4.6 Automate the Authentication Mesh . . . 40
4.6.1 Use-Cases for the Operator . . . 41
4.6.2 Custom Entities for Kubernetes . . . 42
4.6.3 Managing the Public Key Infrastructure . . . 45
4.6.4 Reconciling Authentication Mesh Participants . . . 47

5 Conclusions and Outlook **54**

Bibliography **56**

List of Figures

1 Proof of Concept for heterogeneous authentication . . . 6
2 The Kubernetes Control Loop . . . 10
3 A Sidecar example . . . 12
4 OIDC Authorization Code Flow . . . 14
5 Example of a Trust Zone . . . 15
6 Authentication Mesh Communication Flow . . . 16
7 Use Case Diagram for the PKI . . . 21
8 Prepare CA method in the PKI on Startup . . . 23
9 Deliver public certificate invocation . . . 25
10 Receive Signed Cert from PKI . . . 26
11 Definition of the Common Identity . . . 28
12 Startup Sequence of a Translator . . . 29
13 Encode and Sign a JWT . . . 32
14 Decode JWT and Extract Subject . . . 33
15 Skipped/Ignored Egress Request . . . 34
16 Unauthorized Egress Request . . . 35
17 Forbidden Egress Request . . . 35
18 Processed Egress Request . . . 36
19 Skipped/Ingored Ingress Request . . . 37
20 Errored Ingress Request . . . 37
21 Forbidden Ingress Request . . . 38
22 Successful Ingress Request . . . 39
23 Use-Case Diagram for the Operator . . . 41
24 Custom Resource Definition for a Credential Translator . . . 42
25 Custom Resource Definition for a PKI . . . 43
26 Custom Resource Definition for a Mesh Participant . . . 44
27 Task for the Startup of the Operator to Ensure a PKI . . . 45
28 Reconciliation of a PKI . . . 46
29 Reconcile Mesh Participant - Deployment - Preparation . . . 48
30 Reconcile Mesh Participant - Deployment - Translator Container . . . 49
31 Reconcile Mesh Participant - Deployment - Envoy Configuration . . . 50

32 Reconcile Mesh Participant - Deployment - Envoy Container 51
33 Reconcile Mesh Participant - Deployment - Proxy Environment Variable . 52
34 Reconcile Mesh Participant - Service 53

Declaration of Authorship

I, Christoph Bühler, declare that this project report titled “Common Identities 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 project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the project report 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, March 7, 2022

Christoph Bühler

1 Introduction

"Distributed Authentication Mesh" [1] introduces a theoretical foundation for dynamic authorization in heterogeneous systems. The project, in conjunction with the provided Proof of Concept (PoC) showed, that it is generally possible to transform the identity of a user such that the user can be authorized in another application. In contrast to SAML (Security Assertion Markup Language), the authentication mesh does not require all participants to understand the same authentication and authorization mechanisms. Thus, the mesh is designed to work within a heterogeneous authentication landscape.

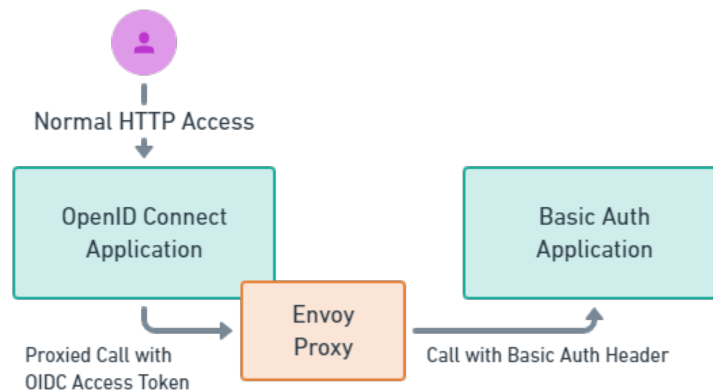

Figure 1: Proof of Concept for heterogeneous authentication

The PoC was designed to show the ability of heterogeneous authentication. Figure 1 shows two applications that were communicating with each other. The source application supports OpenID Connect (OIDC) and the "API" application only supports HTTP Basic authentication. The applications were able to communicate with each other despite the fact, that they do not share the same authentication mechanism. An Envoy proxy enabled the dynamic modification of the HTTP headers. Thus, the PoC did modify the HTTP headers in-flight and replaced the OIDC access token with HTTP Basic credentials.

The project "Distributed Authentication Mesh" mentioned a "common language format" for the transport, but did not define nor implement it [1]. This project enhances the concept of the "Distributed Authentication Mesh" by evaluating and specifying the transport protocol for the common language between services. With this contribution, it will be possible to implement the authentication mesh on a target platform, since the common language is crucial for the success of the mesh. To complement the concept, this work contains an open-source implementation of the mesh for Kubernetes[1]. The

[1]https://kubernetes.io/

implementation is tied to Kubernetes itself, but the concept of the mesh can be adapted to various platforms like Desktop applications.

The remainder of the report describes prerequisites, used technologies with their terminology and further concepts. Section 3, "State of the Authentication Mesh", shows the current version of the concept and which elements are missing. The implementation section shows the concrete evaluation of the common language format combined with the definition and application of the chosen format. Since the implemented version of the mesh runs on Kubernetes, an Operator is created during the implementation to automate the usage of the authentication mesh to allow a good developer experience. The conclusion provides an overview of the results of this work and further gives an outlook for follow-up projects.

2 Definitions and Clarification of the Scope

This section provides general information about the project, the context, and prerequisite knowledge. The scope of the project describes what parts of the additional concepts should be considered. Additionally, this section describes the used technologies (Kubernetes and some specific patterns) for this project and a general overview about secure communication between services.

2.1 Scope of the Project

While the project "Distributed Authentication Mesh" addressed the problem of declarative conversion of user credentials (like an access token from an identity provider) [1], this project focuses on the "common language format" introduced in the former project. This project analyses various variants[2] for such a common language and further implements the common language in Kubernetes. Also, we provide an analysis of various methods to specify and implement such a common language and give an implementation for the selected common format.

As for the implementation of the mesh, this project provides an open-source implementation for the public key infrastructure (PKI) that acts as the trust anchor[3] for the mesh. Furthermore, the evaluated pattern for the common language format is implemented in two different "translators" (HTTP Basic Auth and OpenID Connect). Additionally, an Operator that provides the automation engine of the mesh in context of Kubernetes completes the implementation of the mesh.

Service mesh functions, such as service discovery, are not part of the scope. The authentication mesh should work in conjunction with a service mesh, but does not provide discovery and automated configuration of services. Software that makes use of the authentication mesh must be able to handle the `HTTP_PROXY` and the `HTTPS_PROXY` environment variables to redirect their communication to a forward proxy.

Another topic that is not in the scope of this work, is authentication and authorization of services against the PKI. While there exist mechanisms to authenticate against PKIs, like the usage of a pre-shared key, it is not part of the scope of this project since all participants should reside in the same trust zone. Furthermore, mechanisms such as certificate revocation lists are not implemented in the PKI.

[2] Such as XML, JSON, JWT and so forth.
[3] Trust Anchor: root source of trust for a system, such as a "root certificate" in certificate chains.

2.2 Kubernetes and its Patterns

This section provides knowledge about Kubernetes and two patterns that are used within this project. Kubernetes itself manages workloads and load balances them on several nodes (servers) while the used patterns enable more complex applications and use-cases.

2.2.1 Terminology of Kubernetes

To understand further descriptions and concepts, some core terminology must be understood.

A **Pod** is the smallest possible deployment unit in Kubernetes and contains possible multiple containers. A Pod is defined by a name and a definition for its containers. The containers contain an image (containerized image like a Docker[4] image) and various declarations, such as open ports and environment variables.

A **Deployment** defines the template for a deployed Pod. A deployment defines how a pod should be deployed and how many pods shall run. Furthermore, a deployment manages the update strategy when a new definition of the containing pod is created. This may result in a proper "blue-green deployment" [2], where the new application is started and when it is ready to receive requests, the old one is terminated. There exist multiple deployment specifications, such as `Deployment` and `Stateful Set` which have their own use-cases depending on the specification.

A **Service** makes a Pod (from a deployment) accessible in the Kubernetes world. A service may provide direct access from the outside world or provides an internal DNS address for the Pods. Services may remap exposed ports.

An **Ingress** is a declaration for an entry point to the system. The Ingress points to a service and provides centralized routing from the outside world into some application that runs in Kubernetes. The Ingress may contain definitions for hostnames or path information that are relevant for routing. For an Ingress to function, an `IngressController` must be installed within Kubernetes. The controller is responsible to route traffic to the specified services. Two prominent ingress controllers are "NGINX"[5] and "Ambassador"[6].

2.2.2 Kubernetes, the Orchestrator of Software

Kubernetes is an orchestration software for containerized applications. Originally developed by Google and now supported by the Cloud Native Computing Foundation (CNCF) [3, Ch. 1]. Kubernetes manages the containerized applications and provides access to

[4]https://www.docker.com/
[5]https://www.nginx.com/
[6]https://www.getambassador.io/

applications via “Services” that use a DNS naming system. Applications are described in a declarative way in either YAML or JSON.

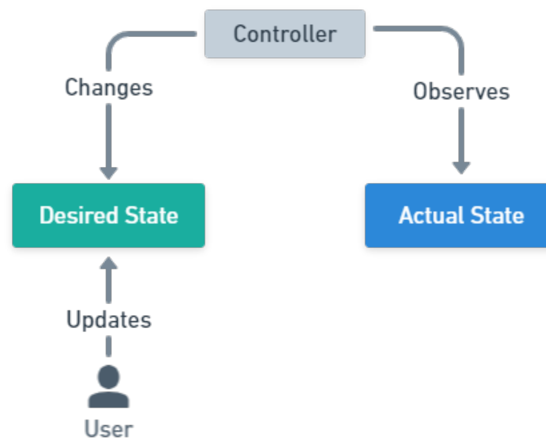

Figure 2: The Kubernetes Control Loop

Figure 2 shows the Kubernetes “control loop”. A controller constantly observes the actual state in the system. When the actual state diverges from the desired one (the one that is “written” in the API in the form of a YAML/JSON declaration) the controller takes action to achieve the desired state. As an example, a deployment has the desired state of two running instances, and currently only one instance is running. The controller will try to start another instance such that the actual state matches the desired one.

2.2.3 An Operator, the Reliability Engineer

The API of Kubernetes is extensible with custom API endpoints (so-called custom resources). With the help of “CustomResourceDefinitions” (CRDs), a user can extend the core API of Kubernetes with their own resources [3, Ch. 16]. An Operator runs in Kubernetes and watches for events on CRDs to manage complex applications. Operators can act as controllers for CRDs with the same loop logic shown in Figure 2.

Site Reliability Engineering (SRE) is a specific software engineering technique to automate software. A team of experts use certain practices and principles to run scalable and highly available applications [4]. Operators are like software for Site Reliability Engineering (SRE). The Operator can automatically manage a database cluster or other complex applications that would require an expert with specific knowledge [5].

Two example operators:

- Prometheus Operator[7]: Manages instances of Prometheus (open-source monitoring and alerting software).
- Postgres Operator[8]: Manages PostgreSQL clusters in Kubernetes.

A partial list of operators available to use is viewable on https://operatorhub.io.

The Prometheus Operator, for example, introduces several CRDs such as `Prometheus`, `ServiceMonitor` and `Alertmanager` [6]. When the Operator is installed into Kubernetes, it reacts to create, update and delete events of `Prometheus` resources. Such a resource could be:

```yaml
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: my-custom-prometheus
spec:
  replicas: 2
  serviceAccountName: prometheus
  serviceMonitorSelector:
    matchLabels:
      foo: bar
```

When the resource above is created or updated in Kubernetes, the Operator will be notified by the Kubernetes API. The Operator then creates a `StatefulSet`[9] that runs the Prometheus Docker image with the scale of two instances (`replicas: 2`). Also, the application will be configured to use a service account named `prometheus` to run in Kubernetes and will automatically search for `ServiceMonitor` resources with a matching label (`foo: bar`) to scrape[10] [6].

Operators can be created by any means that interact with the Kubernetes API. Normally, they are created with some SDK that abstracts some of the more complex topics (like watching the resources and reconnection logic). The following non-exhaustive list shows some frameworks that support Operator development:

- kubebuilder[11]: Go[12] Operator Framework
- KubeOps[13]: .NET Operator SDK
- Operator SDK[14]: SDK that supports Go, Ansible[15] or Helm[16]
- shell-operator[17]: Operator that supports bash scripts as hooks for reconciling

[7] https://github.com/prometheus-operator/prometheus-operator
[8] https://github.com/zalando/postgres-operator
[9] A form of deployment like `Deployment` but with certain stateful mechanics inside Kubernetes.
[10] Scraping: fetch the metrics from the target system and store them with time information
[11] https://book.kubebuilder.io/
[12] https://golang.org/
[13] https://buehler.github.io/dotnet-operator-sdk/
[14] https://operatorframework.io/
[15] https://www.ansible.com/
[16] https://helm.sh/
[17] https://github.com/flant/shell-operator

Operators, and software that implements the Operator pattern, are the most complex extension possibility for Kubernetes, but also the most powerful one [3]. With Operators, whole applications can be automated in a declarative and self-healing way.

2.2.4 A Sidecar, the Extension to a Pod

Sidecars enhance a Pod by injecting additional containers to the defined one [7].

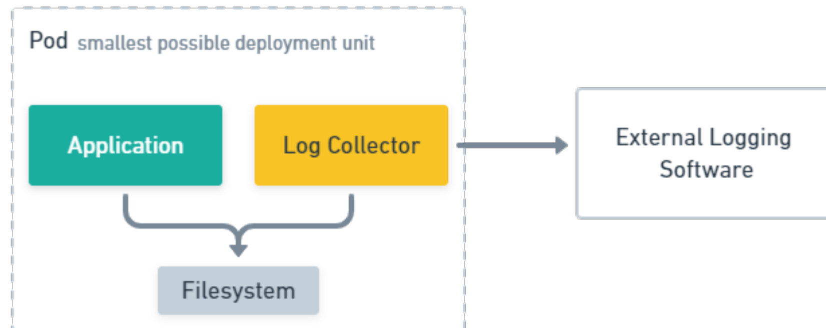

Figure 3: A Sidecar example

Figure 3 shows an example: A containerized application runs in its Docker image and writes logs 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[18].

Sidecars can fulfil multiple use-cases. A service mesh may use sidecars to provide proxies for their 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 Securing Communication

This section provides the required knowledge about security for this project. Authentication and authorization are big topics in software engineering and there exist various standards and mechanisms in the industry. Two of these standards are described below as they are used in this project to show the use-case of the authentication mesh.

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

2.3.1 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 when used 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` [8]. Therefore, the username “test” with the password “high-secret” would result in the header: `Basic dGVzdDpoaWdoLXNlY3JldA==`.

2.3.2 OpenID Connect

OIDC (OpenID Connect) is not specified by an RFC, but by a specification provided by the OpenID Foundation (OIDF). However, OIDC extends OAuth, which in turn is defined by **RFC6749**. OIDC is an authentication scheme that extends `OAuth 2.0`. The OAuth framework only defines the authorization part and how access is granted to data and applications. OAuth, or more specifically the RFC, does not define how the credentials are transmitted [9].

OIDC extends OAuth with authentication, such that it enables login and profile capabilities. OIDC defines three different authentication flows: `Authorization Code Flow`, `Implicit Flow` and the `Hybrid Flow`. These flows specify how the credentials must be transmitted to a server and in which format they return credentials that can be used to authenticate a user [10].

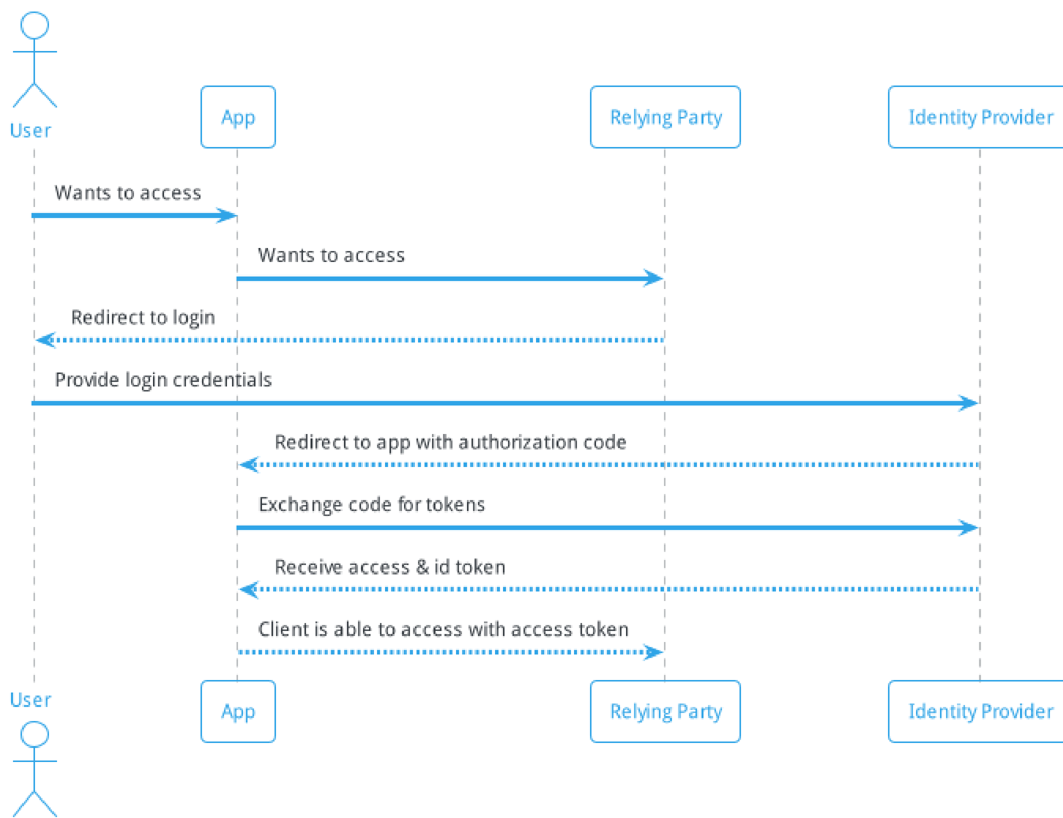

Figure 4: OIDC Authorization Code Flow

As an example, Figure 4 shows a user that 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 Trust Zones and Zero Trust

Trust zones are the areas where software "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. In the same trust zone, other resources can trust the system, that the user has presented valid credentials at some point.

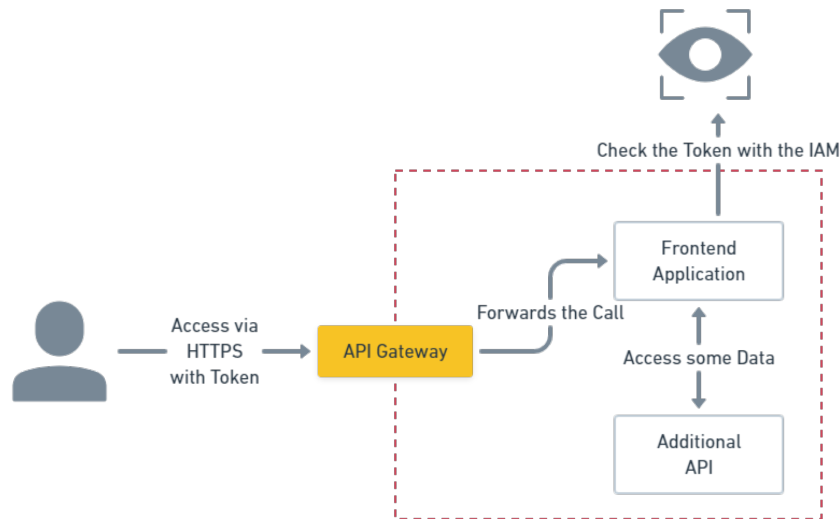

Figure 5: Example of a Trust Zone

As an example, we consider Figure 5. The API gateway is the only way to enter the trust zone. All applications ("Frontend Application" and "Additional API" among others) are shielded from the outside and access is only granted via the gateway. In this scenario, a user first accesses the frontend application and is redirected to the login page. According to the authorization code flow in Figure 4, the frontend application can fetch an access token when the user returns from the login. Then, the user presents his OIDC credentials via HTTP header to the frontend application and the app can verify the token with the IAM (Identity and Access Management) if the credentials are valid. Since the additional API resides in the same trust zone, it does not need to check if the credentials are valid again, the frontend can call the API on the users' behalf.

In contrast to trust zones, "Zero Trust" is a security model that focuses on protecting (sensitive) data [11]. Zero trust assumes that every call could be intercepted by an attacker. Therefore, all requests must be validated. As a consequence, the frontend in Figure 5 is required to send the user token along with the request to the API and the API checks the token again for its validity. For the concept of zero trust, it is irrelevant if the application resides in an enterprise network or if it is publicly accessible.

A concern to address, in zero trust or authentication in general, is the authentication of the authenticator. Who assures that, in the given example, the IAM is not a corrupted instance that allows attackers to inject faulty information? Such authentication software is hardened and developed over several months and years. It is not possible to create the perfect safe application. But a partial solution is to use well-known software[19] and applications to provide the safest possible implementations of such software.

[19]For example "Auth0", "Keycloak" or "Octa" among others.

3 State of the Authentication Mesh

This section shows the deficiencies that this project tries to solve. Since this project enhances the concepts of the "Distributed Authentication Mesh", many elements are already defined in the past work.

Common Language Format for Communication

The "Distributed Authentication Mesh" defines an architecture that enables dynamic conversion of user identities in a declarative way [1]. The common language format however, is neither defined nor implemented yet. Past work did implement a Proof of Concept (PoC) to show the general idea, but did not prove the feasibility with the common language format. To enable the creation of a production-grade software based on the concepts of the authentication mesh, the common language must be defined and specified.

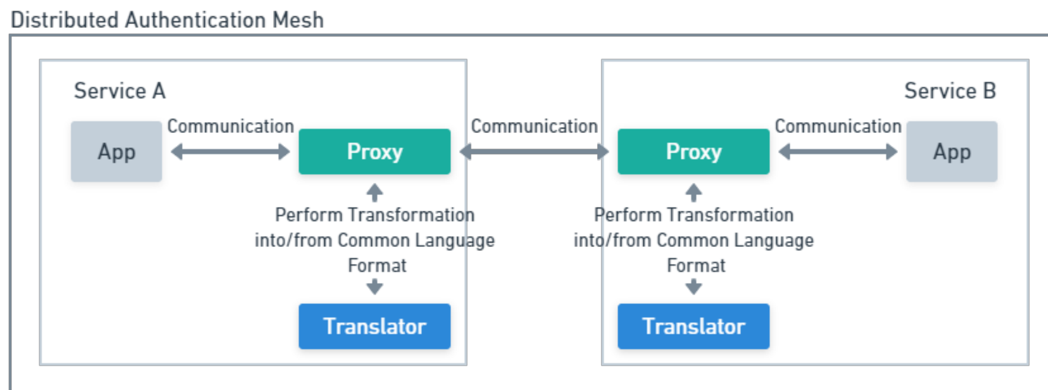

Figure 6: General communication flow of two services in the distributed authentication mesh

Figure 6 shows the communication between two services that are part of the distributed authentication mesh. The communication of the app in service A is proxied and forwarded to the translator. The translator then determines if the request contains any relevant authentication information. Then the translator converts the information into a common language format that the translator of service B understands. After this step, the proxy forwards the communication to service B. The proxy in service B will recognize the custom language format in the HTTP headers and uses its transformer to create valid credentials (such as username/password) out of the custom language format, such that the app of service B can authenticate the user. If service A and B use the same authentication

scheme, this transformation is not needed. However, if a heterogeneous authentication landscape is present, where service A uses OIDC and service B is a legacy application that only supports Basic Auth, the need for transformation arises.

The mentioned common language format is not specified. This project analyzes various forms of such a common language and specifies the language along with the requirements. Furthermore, an implementation shall be provided for Kubernetes to see the concepts in a productive environment.

4 Implementing a Common Language and a Mesh Operator

This section analyzes different approaches to utilize a common language format between the services of the "Distributed Authentication Mesh". After the analysis, the definition and implementation of the common format enhances the general concept of the Mesh and enables a production-grade software. The PKI and the translators are written in Go, while the Kubernetes Operator is written in C#. All software is licensed under the **Apache-2.0** license and can be found in the "WirePact" organization on GitHub[20].

4.1 Goals and Non-Goals of the Project

As mentioned, this project enhances the concept of the distributed authentication mesh by analyzing various ways of transmitting the user identity and defining a meaningful way to transport the identity between participants. As such, the goals and non-goals of this project remain the same as in the past work of the distributed authentication mesh [1, Ch. 4].

Additional **functional requirements**:

- The translator generates/parses the common language format.
- The translator is able to validate the integrity of the transmitted identity.

Additional **non-functional requirements**:

- The common language contains all needed information to identify a user.
- The translator acts as proxy for the services behind the mesh. This ensures that requests can be intercepted for applications that are part of the mesh.
- The usage of the authentication mesh shall provide a good developer experience.

The two lists above extend the existing requirements from the past work in [1]. In general, the system must not be less secure than the current existing security standards. The definition of the common language format must contain a way to check the integrity of the transmitted data and the translators must not interfere with the data stream and must only modify HTTP headers.

4.2 A Way to Communicate with Integrity

To enable the translators in the distributed authentication mesh to communicate securely, a common format must be used [1]. The format must support a feasible way to prevent modification of the data it transports. The following sections give an overview over the three options that may be used. In the end of the section a comparison shows pro and contra to each option and a decision towards a format is made.

[20] https://github.com/WirePact

4.2.1 YAML, XML, JSON, and Others

YAML (YAML Ain't Markup Language[21]), XML (Extensible Markup Language[22]), JSON (JavaScript Object Notation[23]) and other structured data formats such as binary serialized objects in C# are widely used for data transport. They are typically used to transport structured data in a more or less human-readable form but maintain the possibility to be serialized and deserialized by a machine. Those structures could be used to transport the identity of an authenticated user. However, the formats do not support integrity checks by default.

```yaml
userId: 123456
userName: Test User
```

The example above shows a simple YAML example with an "object" that contains two properties: `userId` and `userName`. These objects can be extended and well typed in programming languages.

There exist approaches like "SecJSON" that enhance JSON with security mechanisms such as data encryption [12]. But if the standard of the specification is used, no integrity check can be performed and the translators of the authentication mesh cannot detect if the data was modified. Thus, using a "simple" structured data format for the transmission of the user identity would not suffice the security requirements of the system.

Similar to "SecJSON", one could add special fields into XML and/or YAML to transmit a hash of the data such that the receiver can validate the data. However, using custom fields does not rely on current standards and are therefore prone to errors implementation wise.

4.2.2 X509 Certificates

The x509 standard (**RFC5280**) defines how certificates shall be used. Today, the connection over HTTPS is done via TLS and certificate encryption. The fields in a certificate are only partially defined. These "standard extensions" are well-known fields such as the "authority" or alternative names for the subject. In the specification, "private extensions" are another possibility to encode data into certificates [13, Ch. 4]. These extensions could be used to transmit the data needed for the distributed authentication mesh.

Certificates have a big advantage: they can be integrity checked via already implemented hashing mechanisms and provide a "trust anchor"[24] in the form of a root certificate authority (root CA). Furthermore, if certificates would be used to transmit the users'

[21] https://yaml.org/
[22] https://www.w3.org/XML/
[23] https://www.json.org/
[24] A trust anchor is a root for all trust in the system.

identity within the authentication mesh, the certificates could also be used to harden the communication between two services. The certificates can enable mutual TLS (mTLS) between communicating services.

But, implementing custom private fields and manipulating that data is cumbersome in various programming languages. In C# for example, the code to create a simple x509 certificate can span several hundred lines of code. Go[25] on the other hand, has a much better support for manipulating x509 certificates. Since the result of this project should provide a good developer experience, using x509 certificates is not be the best solution to solve the communication and integrity issue. If future work implements mTLS to harden the communication between services, it may be feasible to transmit the users' identity within the used certificates.

4.2.3 JSON Web Tokens

A plausible way to encode and protect data is to encode them into a JSON web token (JWT). JWTs are used to encode the user identity in OpenID Connect and OAuth 2.0. A JWT contains three parts: A "header", a "payload" and the "signature" [14]. The header identifies which algorithm was used to sign the JWT and can contain other arbitrary information. The payload carries the data that shall be transmitted. The last part of the JWT contains the constructed signature of the header and the payload. This signature is constructed by either a symmetrical or asymmetrical hashing algorithm.

To make use of JWTs in the distributed authentication mesh, another technique for JWTs is used: JSON Web Signatures (JWS). JWS represents data that is secured with a digital signature [15]. When considering JWT/JWS for the mesh, a signed token that contains the user ID could be used with key material from the PKI to sign the data and provide a way to securely transmit the data. Since the data is not confidential (typically only a user ID), it must be signed only. To help prevent extra round trips, the two extra headers `x5c` and `x5t` can be used to transmit the certificate chain as well as a hash of the certificate to the proxy that is checking the data [15].

In contrast to the above-mentioned "SecJSON", a JWT is well-defined by an RFC. SecJSON enables encrypted data within JSON but does lack the means of integrity checking. A JWT does not encrypt the data but uses JWS for hashing and signing of the data to prevent modification of the data.

4.2.4 Using JWT in the Authentication Mesh

After considering the possible transport formats above, we can now analyze the pro and contra arguments. While **structured formats** like YAML and JSON are widely known and easily implemented, they do not offer a built-in mechanism to prevent data

[25] https://go.dev/

manipulation and integrity checking. There are standards that mitigate that matter, but then one can directly use JWT instead of implementing the mechanism by themselves.

X509 certificates provide an optimal mechanism to transmit extra data with the certificate itself with "private extensions". They could also be used to enable mTLS between services to further harden the communication between participants of the mesh. However, to enable developers to implement custom translators by themselves, x509 certificates are not optimal since the code to manipulate them heavily depends on the used programming language.

JWT uses the best of both worlds. They are asynchronously signed with a x509 certificate private key and can transmit the certificate chain as well as a hash of the signing certificate to prevent manipulation. There exist various libraries for many programming languages like Java, C# or Go. Also, JWTs are already used in similar situations like the ID tokens for OpenID Connect.

4.3 A Public Key Infrastructure as Trust Anchor

The implementation of a PKI is vital to the authentication mesh. The participating translators must be able to fetch valid certificates to sign the JWTs they are transmitting. The PKI can be found at: https://github.com/WirePact/k8s-pki. The PKI must fulfill the use cases depicted in Figure 7.

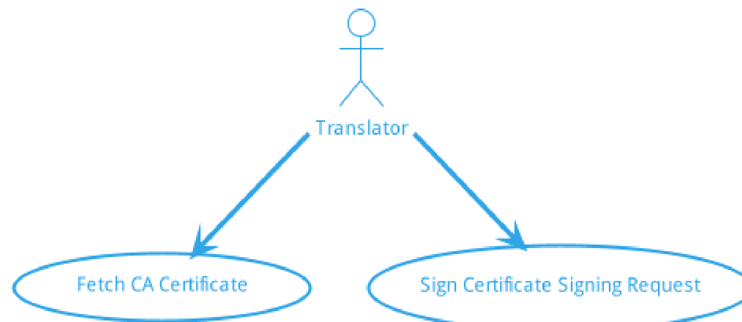

Figure 7: Use Case Diagram for the PKI

Fetch CA Certificate. Any translator must have access to the root CA (certificate authority) to validate the signatures of received JWTs. The signing certificates of the translators are derived by the CA and can therefore be validated if they are authorized to be part of the mesh.

Sign Certificate Signing Requests. The participating clients (translators) must be able to create a certificate signing request (CSR) and send them to the PKI. The PKI does create valid certificates that are signed with the root CA and then returns the

created certificates to the clients. To validate the certificate chain, an interested party can fetch the public part of the root CA via the other mentioned endpoint and check if the chain is valid.

4.3.1 "Gin", a Go HTTP Framework

To manage HTTP request to the PKI, the "Gin"[26] framework is used. It allows easy management of routing and provides support for middlewares if needed. To set up the CA, the following code enables a web server with the needed routes to `/ca` and `/csr`:

```
router := gin.Default()

router.GET("ca", api.GetCA)
router.POST("csr", api.HandleCSR)
```

4.3.2 Prepare the CA

Since the implementation targets a local environment (for development) as well as the Kubernetes environment, the CA and the private key can be stored in multiple ways. For local development, the certificate with the private key is created in the local file system. If the PKI runs within Kubernetes, a `Secret` (encrypted data in Kubernetes) shall be used.

[26] https://github.com/gin-gonic/gin

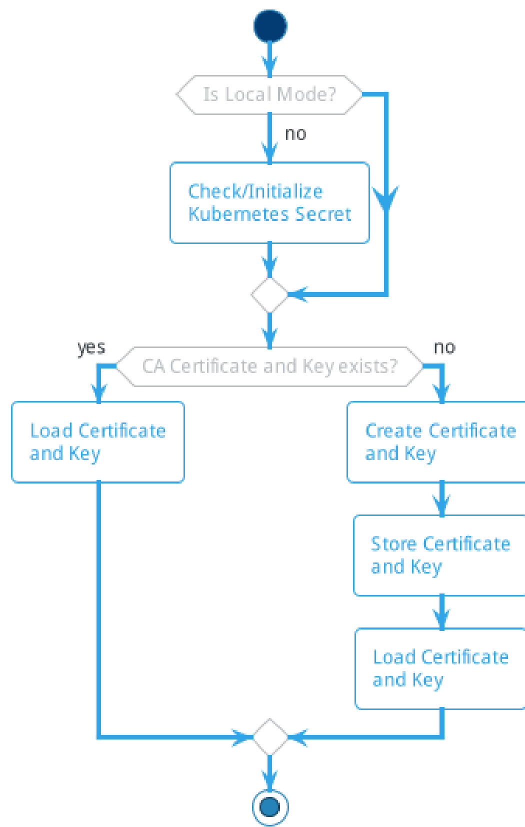

Figure 8: Prepare CA method in the PKI on Startup

During the startup of the PKI, a “prepare CA” method runs and checks if all needed objects are in place. Figure 8 shows the invocation sequence of the method. If the PKI runs in “local mode” (meaning it is used for local development and has no access to Kubernetes), the CA certificate and private key are stored in the local file system. Otherwise, the Kubernetes secret is prepared and the certificate loaded/created from the secret.

To create a new CA certificate, the following code can be used:

```
ca := &x509.Certificate{
    SerialNumber: big.NewInt(getNextSerialnumber()),
    Subject: pkix.Name{
        Organization: []string{"WirePact PKI CA"},
        Country:      []string{"Kubernetes"},
        CommonName:   "PKI",
    },
    NotBefore: time.Now(),
    NotAfter:  time.Now().AddDate(20, 0, 0),
    IsCA:      true,
```

```
    ExtKeyUsage: []x509.ExtKeyUsage{
        x509.ExtKeyUsageClientAuth,
        x509.ExtKeyUsageServerAuth,
    },
    KeyUsage: x509.KeyUsageDigitalSignature |
        x509.KeyUsageCertSign,
    BasicConstraintsValid: true,
}

privateKey, _ := rsa.GenerateKey(rand.Reader, 2048)
publicKey := &privateKey.PublicKey
caCert, err := x509.CreateCertificate(
    rand.Reader,
    ca,
    ca,
    publicKey,
    privateKey)
```

The private key is generated with a cryptographically secure random number generator. After the certificate is generated, it can be encoded and stored in a file or the Kubernetes secret. The CA certificate is created with a 20-year lifetime. A further improvement to the system could introduce short-lived certificates to mitigate attacks against the CA.

4.3.3 Deliver the CA

As soon as the preparation process in Figure 8 has finished, the CA certificate is ready to be delivered in-memory. This process does not need any special processing power. When a `HTTP GET` request to `/ca` arrives, the PKI will return the public certificate part of the root CA to the caller. This call is used by translators and other participants of the authentication mesh to store the currently valid root CA by themselves and to validate the certificate chain.

```
context.Header(
    "Content-Disposition",
    `attachment; filename="ca-cert.crt"`)
context.Data(
    http.StatusOK,
    "application/x-x509-ca-cert",
    certificates.GetCA())
```

The only specialty is the data type headers that are set to `application/x-x509-ca-cert`. While they are not necessary, the headers are added for good practice.

The `GetCA()` method itself just returns the public CA certificate:

```
func GetCA() []byte {
    return pem.EncodeToMemory(
```

```
        &pem.Block{Type: "CERTIFICATE", Bytes: ca.Raw})
}
```

The certificates are “PEM”[27] encoded.

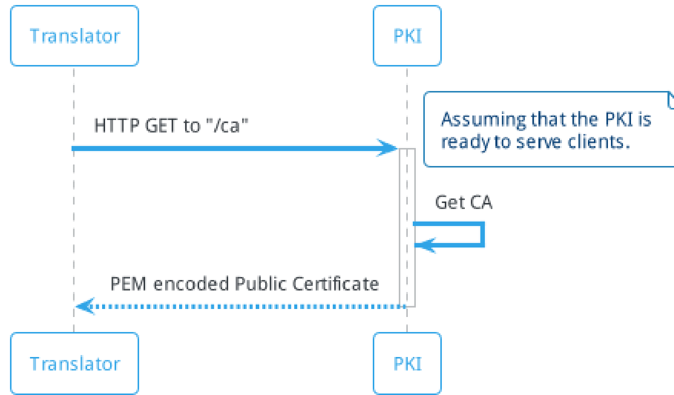

Figure 9: Deliver public certificate invocation

The process to deliver the CA is not very complex, as is shown in Figure 9. As soon as the startup process in Figure 8 is finished, the PKI can return the public part of the certificate to any client that sends a `HTTP GET` to `/ca` of the PKI.

4.3.4 Process Certificate Signing Requests (CSR)

To be able to sign CSRs, as stated in Figure 7, the PKI must be able to parse and understand CSRs. The PKI supports a `HTTP POST` request to `/csr` that receives a body that contains a CSR.

[27] PEM Encoding: https://de.wikipedia.org/wiki/Privacy_Enhanced_Mail

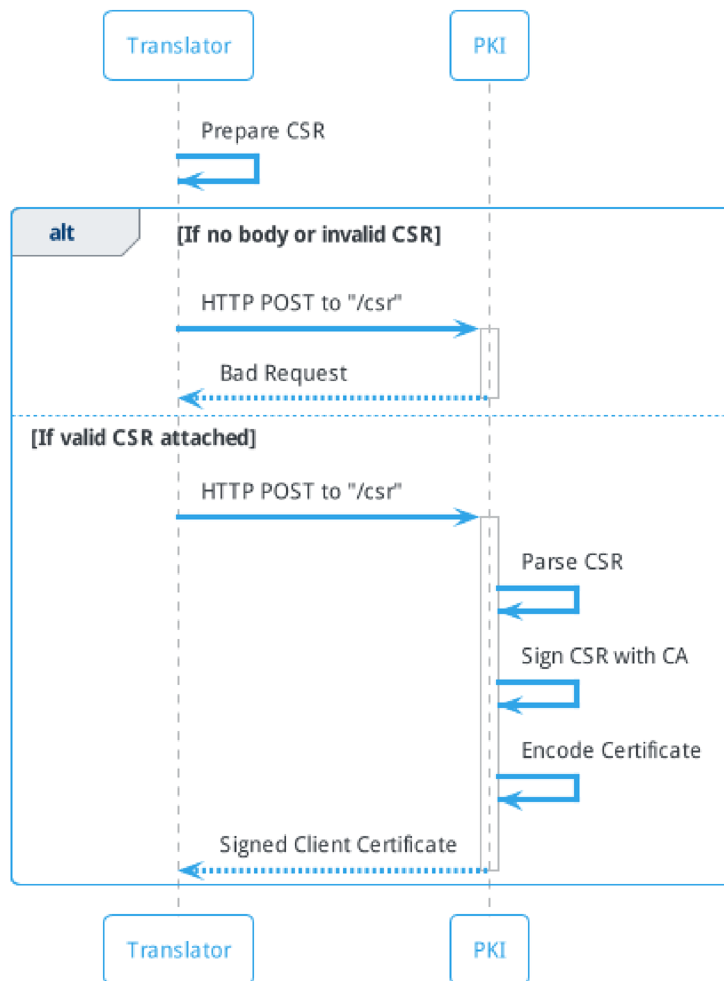

Figure 10: Invocation sequence to receive a signed certificate from the PKI.

The sequence in Figure 10 runs in the PKI. Since the certificate signing request is prepared with the private key of the translator (or any other participant of the mesh), no additional keys must be created. The PKI signs the CSR and returns the valid client certificate to the caller. The caller can now sign data with the private key of the certificate and any receiver is able to validate the integrity with the public part of the certificate. Furthermore, the receiver of data can validate the certificate chain with the root CA from the PKI.

If no CSR is attached to the `HTTP POST` call, or if the body contains an invalid CSR, the PKI will return a `HTTP Bad Request` (status code 400) to the sender and abort the call.

4.3.5 Authentication and Authorization against the PKI

In the current implementation, no authentication and authorization against the PKI exists. Since the current state of the system shall run within the same trust zone, this is not a big threat vector. However, to truly achieve a *distributed* authentication mesh, a mechanism to create trust between several trust zones must be implemented.

A security consideration in the distributed authentication mesh is the possibility that *any* client can fetch a valid certificate from the PKI and then sign *any* identity within the system. To harden the PKI against unwanted clients, two possible actions can be taken.

Use a pre-shared key to authorize valid participants. With a pre-shared key, all valid participants have the proof of knowledge about something that an attacker should have a hard time to come by. In Kubernetes this could be done with an injected secret or a vault software[28].

Use an intermediate certificate for the PKI. When the PKI itself is not the absolute trust anchor (the root CA), an intermediate certificate could be delivered as a pre-known secret. Participants would then sign their CSRs with that intermediate certificate and therefore proof that they are valid participants of the mesh.

In either way, both options require the usage of pre-shared or pre-known secrets. Additional options to mitigate this attack vector are not part of this project and shall be investigated in future work.

4.4 Provide a Translator Base

To enable developers to create translators easily, the GitHub organization "WirePact"[29] provides a translator base package written in Go. This package contains helpers and utilities that are needed in a translator and further provide a developer friendly way to implement a translator. The package is available on the GitHub repository https://github.com/WirePact/go-translator.

4.4.1 Define the Common Identity

The distributed authentication mesh needs a single source of truth. It is not possible to recover user information out of arbitrary information. As an example, an application that uses multiple services with OIDC and Basic Auth needs a common "base of users". Even if the authentication mesh is not in place, the services need to know which basic authentication credentials they need to use for a specific user.

[28]Like "HashiVault" https://www.vaultproject.io/

[29]WirePact: The development name for the distributed authentication mesh.

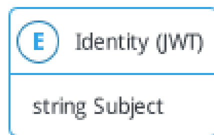

Figure 11: Definition of the Common Identity

As shown in Figure 11, the definition of the common identity is quite simple. The only field that needs to be transmitted is the `subject` of a user. The `subject` (or `sub`) field is defined in the official public claims of a JWT [14, Sec. 4.1.2]. Any additional information that is provided may or may not be used. Since the system is designed to work in a heterogeneous landscape, the common denominator is the users' ID. For some destinations, additional information could be helpful, but it is not guaranteed that the information is available at the source.

4.4.2 Startup a Translator

When a translator is created with the `NewTranslator()` function of the translator package, a struct type is instantiated that provides some utility functions. Upon creation, the new translators contains two web-servers with the configured ingress and egress ports. Those web-servers are configured to listen to gRPC calls from envoy. The servers in the translator are created but not yet started. They are ready to be run.

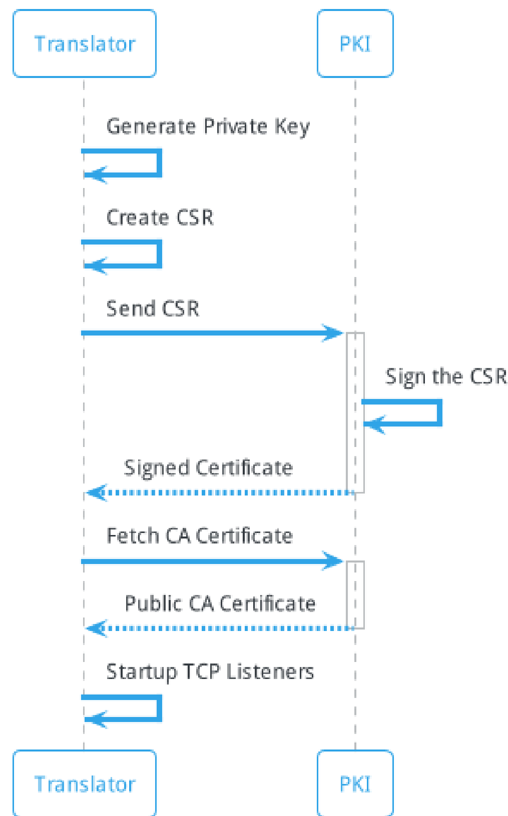

Figure 12: Startup Sequence of a Translator

Figure 12 shows the startup sequence for a translator that was created with the provided package. As soon as the translator gets started (with `translator.Start()`), the translator first ensures its own key material. This means, that a private key for the local certificate is generated if it does not exist. Further, if the local certificate does not exist, a certificate signing request (CSR) is created and sent to the PKI. Upon success, the PKI returns a valid and signed certificate that the translator can use to sign the JWTs. The last preparation step is to fetch the public part of the CA certificate from the PKI to validate incoming JWTs.

When the preparations for the key material are done, two go routines start the web-servers (listeners) for incoming and outgoing request authentication.

```
go func() {
    logrus.Info("Serving Ingress")
    err := translator.
        ingressServer.
        Serve(*translator.ingressListen)
    if err != nil {
```

```
        logrus.
        WithError(err).
        Fatal("Could not serve ingress.")
    }
}()
```

These listeners are now ready to receive gRPC calls from Envoy. Envoy must be configured to send an authentication check for all intercepted incoming and outgoing calls to the respective destination. The translator now awaits an interrupt, terminate or kill signal to gracefully shut down the listeners.

4.4.3 Provide Endpoints for Interception

On top of the mentioned listeners are two wrapper methods. A developer provides the ingress and egress function to the library, which in turn then encapsulates the functions with the effective gRPC call from Envoy. When creating a translator, the provided egress function just receives the `CheckRequest` and the ingress function receives a `string` for the parsed subject (user ID) and the `CheckRequest` from Envoy as parameters. They return their respective result (`IngressResult` and `EgressResult`) which then decides the fate of the request.

```
result, err := server.EgressTranslator(req)
if err != nil {
    return nil, err
}

if result.Skip {
    return envoy.CreateNoopOKResponse(), nil
}

if result.UserID == "" {
    return envoy.CreateForbiddenResponse(
        "No UserID given for outbound communication."
    ), nil
}

if result.Forbidden != "" {
    return envoy.CreateForbiddenResponse(result.Forbidden),
        nil
}

return envoy.CreateEgressOKResponse(
    server.JWTConfig,
    result.UserID,
    result.HeadersToRemove
)
```

The function above shows the logic for outgoing (egress) communication. The developer provides the `server.EgressTranslator(req)` implementation at the start of the translator. The package then in turn calls this function and handles the result according to the logic above.

```
wirePactJWT, ok := req.Attributes.
    Request.Http.
    Headers[wirepact.IdentityHeader]
if !ok {
    return envoy.CreateNoopOKResponse(), nil
}

subject, err := wirepact.GetJWTUserSubject(wirePactJWT)
if err != nil {
    return nil, err
}

result, err := server.IngressTranslator(subject, req)
if err != nil {
    return nil, err
}

if result.Skip {
    return envoy.CreateNoopOKResponse(), nil
}

if result.Forbidden != "" {
    return envoy.CreateForbiddenResponse(
        result.Forbidden
    ), nil
}

return envoy.CreateIngressOKResponse(
    result.HeadersToAdd,
    append(result.HeadersToRemove, wirepact.IdentityHeader)
), nil
```

On the other hand, incoming (ingress) communication requires an extra step. First, a check ensures that the authentication mesh header is present. If not, the request gets forwarded to the destination without any interruption. If a JWT is present, the JWT is decoded and the subject (user ID) extracted. The next step involves the provided `server.IngressTranslator` from the developer that coded the translator. The last step is similar to egress communication, where the result of the translator function is parsed and executed accordingly.

4.4.4 Encode the JWT

Since the chosen technology to transport the users' identity is a JSON web token, the package provides a simple way to en-, and decode the JWTs. One thing that must be provided to the JWT is the subject (i.e. the users ID). Since we defined that the only required thing for the identity is the user ID (see Figure 11), we encode it in the official specified JWT way and name it "sub" (i.e. "subject").

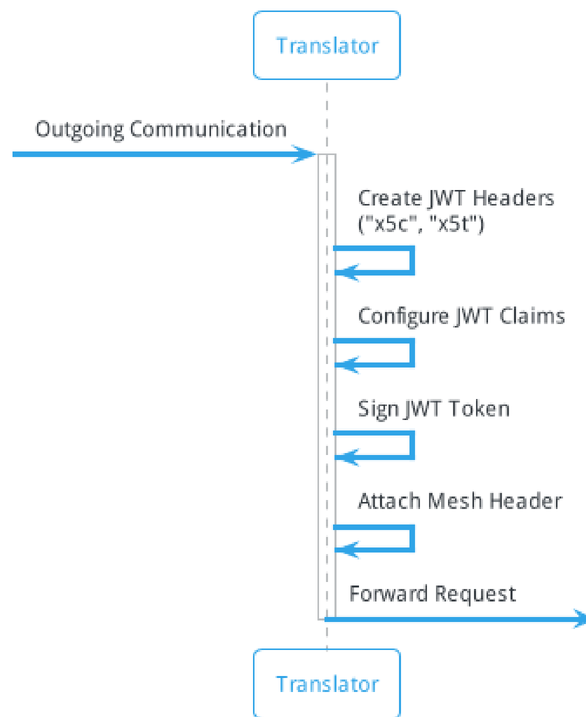

Figure 13: Encode and Sign a JWT

The logic in Figure 13 shows what happens to a user ID if a JWT shall be created. The JWT headers (`x5c` and `x5t`) are created from the local certificate chain and the fetched CA certificate from the PKI. Those headers are used by the receiving party to check if the JWT is valid and from a participant of the authentication mesh. Next, the JWT claims are configured (subject, issuer, expiry date and so forth) and the token is signed. The signed token is then injected into the HTTP call with a special `x-wirepact-identity` header.

4.4.5 Decode the JWT

On the receiving side, the process is reversed.

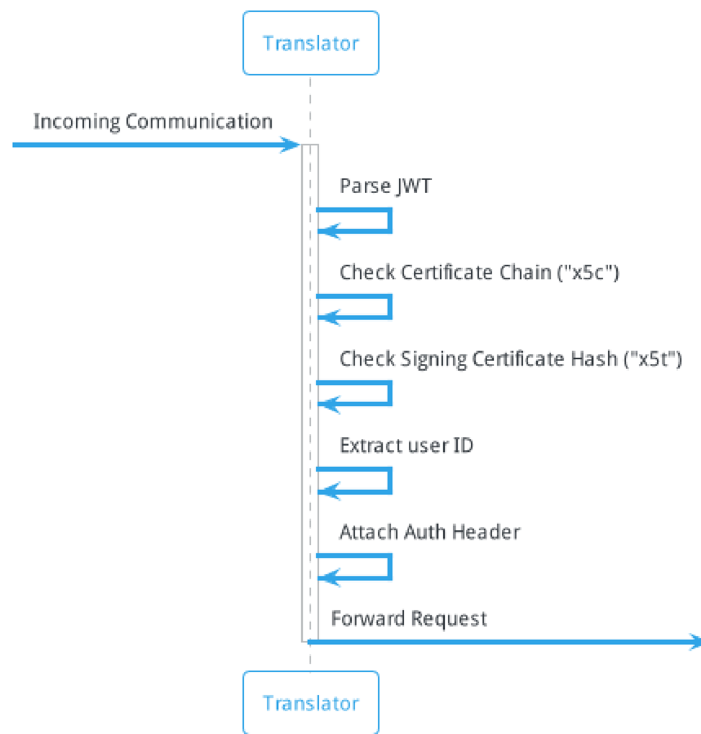

Figure 14: Decode JWT and Extract Subject

Figure 14 shows the process for incoming communication. When the translator receives a call from the outside world that contains the mesh HTTP header (`x-wirepact-identity`), then the process runs. First, the encoded header data is parsed as JWT. Next, the translator checks, if the encoded certificate chain (`x5c`) is valid and matches its own CA certificate. Then the attached certificate hash (`x5t`) is checked against the signing certificate of the source (which must be the first certificate in the provided certificate chain). The headers that are checked are defined in the JWT specification [15]. If a subject can be extracted, the developers code will be called and in the end, the request is forwarded if everything went fine.

4.5 Implementing an HTTP Basic Translator with a Secure Common Identity

This section describes the usage of the secure common identity mentioned above within a translator. The translator uses HTTP Basic Auth (RFC7617 [8]) for the username/password combination. The implementation is hosted on https://github.com/WirePact/k8s-basic-auth-translator.

4.5.1 Validate and Encode Outgoing Credentials

Any application that shall be part of the authentication mesh must either call the injected forward proxy by itself, or it should respect the HTTP_PROXY environment variable, as done by Go and other languages/frameworks. Outgoing communication ("egress") is processed by the envoy proxy with the external authentication mechanism [1]. The following results exist:

- Skip: Do not process any headers or elements in the request
- UserID empty: forbidden request
- Forbidden not empty: for some reason, the request is forbidden
- UserID not empty: the request is allowed

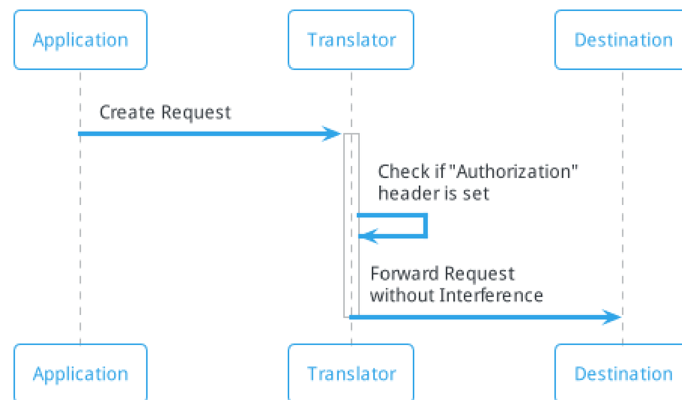

Figure 15: Skipped/Ignored Egress Request

Figure 15 shows the sequence if the request is "skipped". In this case, skipped means that no headers are consumed nor added. The request is just passed to the destination without any interference. This happens if, in the case of Basic Auth, no HTTP Authorize header is added to the request or if another authentication scheme is used (OIDC for example). The possibility to skip a request enables front-facing applications to still receive normal requests that do not contain any authentication information. As an example, this can happen when an application periodically calls some service that does not need any credentials. The neutral request must not be rejected or forbidden.

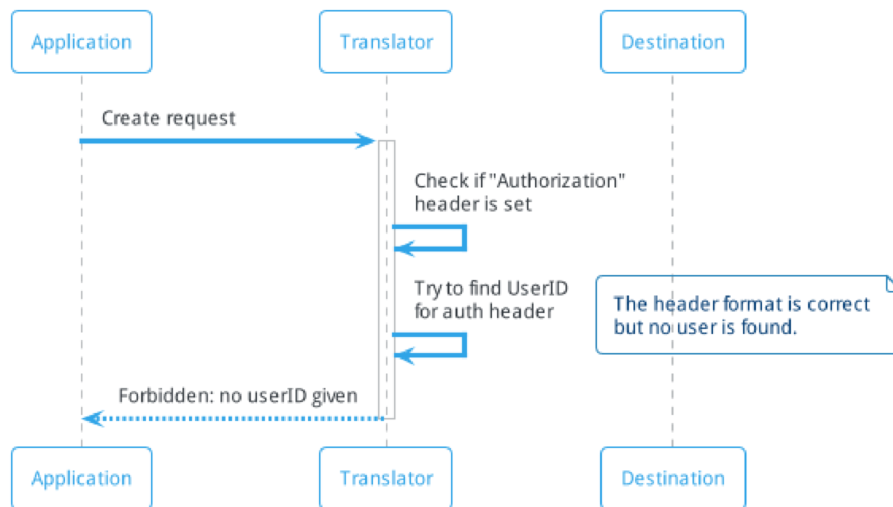

Figure 16: Unauthorized Egress Request

Figure 16 depicts the process when the request contains a correct HTTP header, but the provided username/password combination is not found in the “repository” of the translator. So, no common user ID can be found for the given username and therefore, the provided authentication information is not valid.

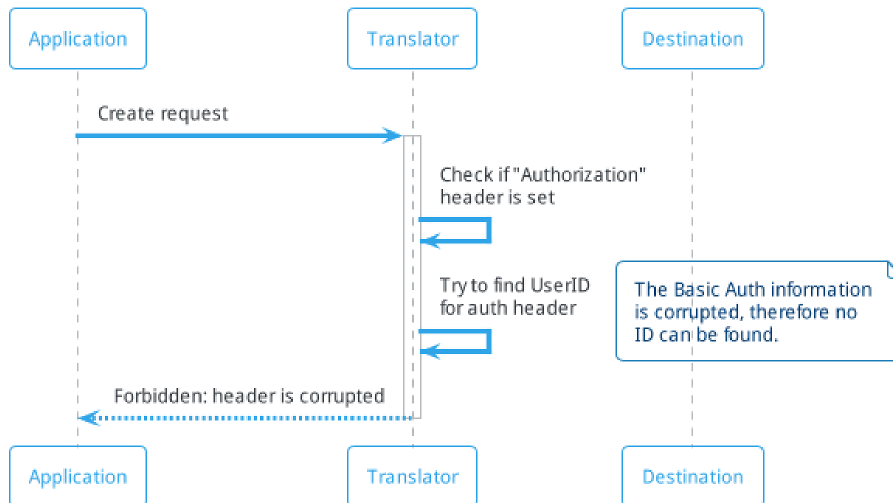

Figure 17: Forbidden Egress Request

In Figure 17, the HTTP header is present, but corrupted. For example, if the username/password combination was encoded in the wrong format. If this happens, the proxy

will reject the request and never bother the destination with incorrect authentication information.

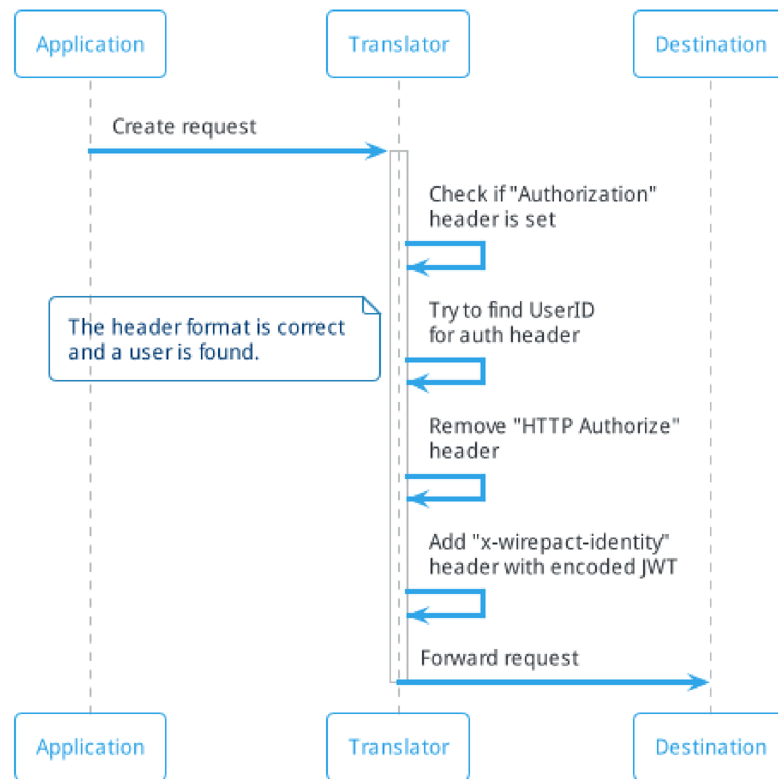

Figure 18: Processed Egress Request

If a request contains the correct HTTP header, the data within is valid and a user can be found with the username/password combination, Figure 18 shows the process of the request. The translator instructs the forward proxy to consume (i.e. remove) the HTTP authorize header and injects a new custom HTTP header `x-wirepact-identity`. The new header contains a signed JWT that contains the user ID as the subject and the certificate chains as well as a hash of the signing certificate in its headers.

4.5.2 Validate and Decode an Incoming Identity

The transformer also intercepts incoming connections via the external authentication feature of envoy. If a call contains the specified HTTP header (`x-wirepact-identity`) that contains a JWT, the translator tries to validate the information. In general, there exist four different reactions of the translator:

- Skip: if no information is given that relates to the authentication mesh.

- Error: if any error happens during validation.
- Forbidden: if the request is forbidden for any reason.
- OK: if the request is valid and the information about the user could be gathered.

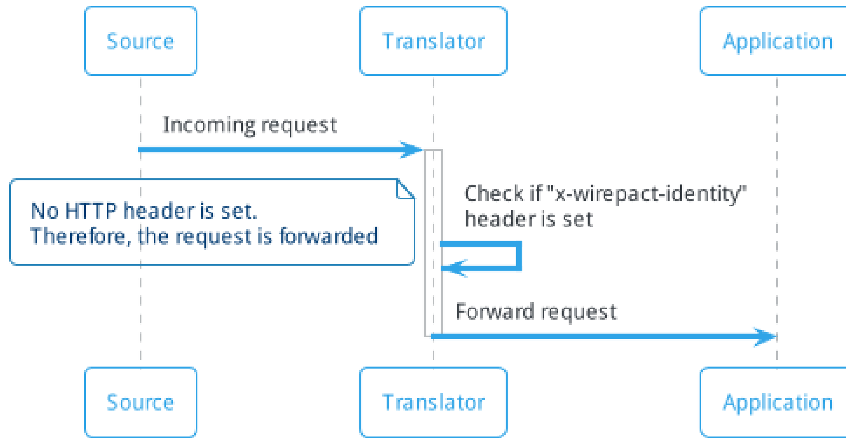

Figure 19: Skipped/Ingored Ingress Request

In Figure 19, the process for a neutral request is shown. The request contains no specific information that is relevant for the authentication mesh. Since the translator may not interfere with requests that are not "part of the mesh", the request is skipped. The destination application may handle the request appropriately. As an example, the target application can request the source of the request to sign in. This process allows normal requests to be handled in the mesh. If all requests without mesh information would be blocked, no "normal" request could be sent.

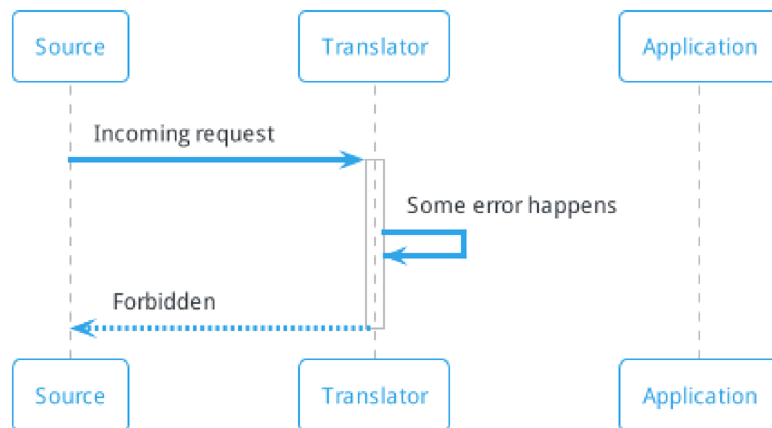

Figure 20: Errored Ingress Request

Error handling in the translator is bound to return forbidden responses. The translator should not throw any errors if possible [1]. But if there are some errors, the translator returns a forbidden request to deny access to the destination as seen in Figure 20. If the translator would just skip the request, this could make the system vulnerable against error attacks, where an attacker could force some error to happen in the translator and then reach the destination.

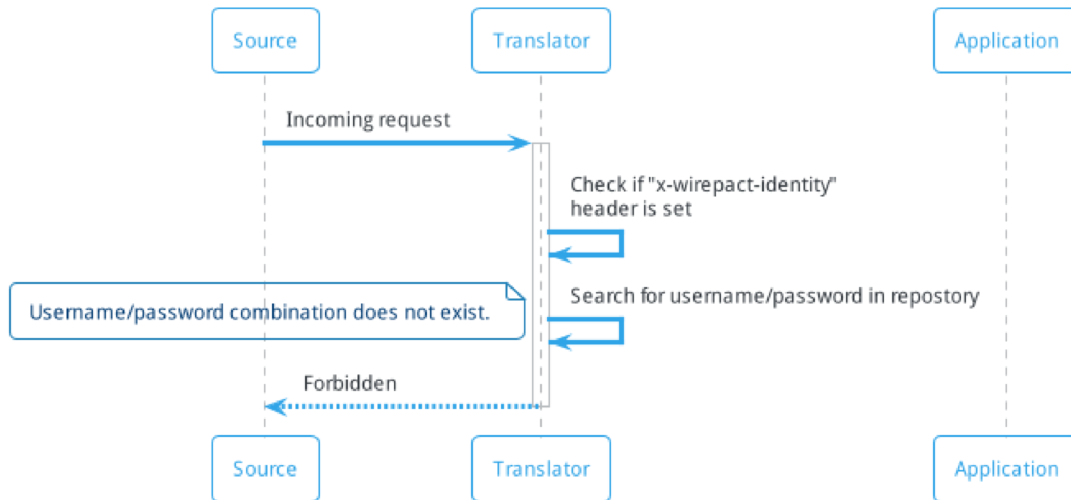

Figure 21: Forbidden Ingress Request

If the incoming request contains an `x-wirepact-identity` HTTP header and the subject of the user could be extracted successfully, the translator searches for a username/password combination in its repository. If no credentials are found, as shown in Figure 21, the request is denied. No valid credentials mean that the translator cannot attach valid basic credentials for the target system.

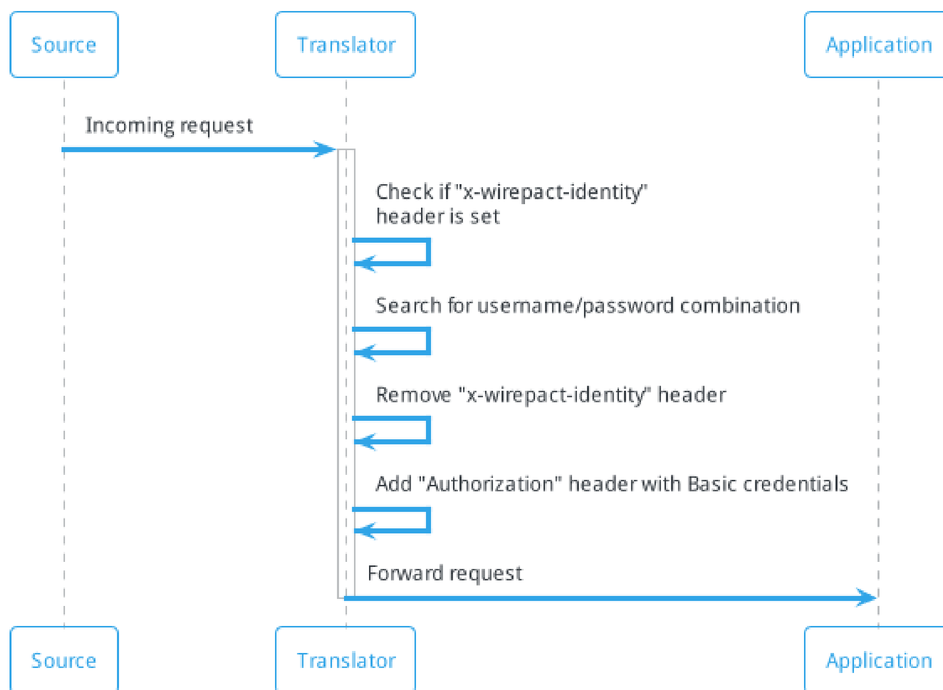

Figure 22: Successful Ingress Request

In contrast to the situations above, Figure 22 shows the successful request. If the subject could be parsed, validated and there exists a proper username/password combination in the translators' repository, the translator instructs envoy to consume (i.e. remove) the artificial mesh header and attach the basic authentication header for the target system. In this case, the target system receives valid credentials that it can validate despite the fact that the original source may not have used basic authentication.

4.5.3 Contrast to an OIDC Translator

Since the HTTP Basic translator mentioned above has a common base with other translators, any other authentication/authorization mechanism can be programmed into a translator. As a further example, and to demonstrate the feasibility of the solution, another translator that handles OpenID Connect (OIDC) was created. The translator resides on GitHub in the repository https://github.com/WirePact/k8s-keycloak-oidc-translator.

The OIDC translator is specifically implemented to work in conjunction with a "Keycloak"[30] instance. Keycloak is an open-source identity and access management (IAM)

[30] https://www.keycloak.org/

system that provides the necessary configuration interface to easily use OIDC within your system architecture.

The differences of the OIDC translator to the HTTP Basic Translator are as follows:

- Other configuration (`ISSUER`, `CLIENT_ID`, and `CLIENT_SECRET`) needed
- Reacts to `Authorization: Bearer ...` headers
- Fetches user access token via token exchange[31]

The basic logic of the translator remains the same. If an outgoing request contains an authorization HTTP header, the header will be consumed. The access token is validated against Keycloak and if it returns a subject (i.e. user-ID) via the introspection endpoint of Keycloak, the ID is encoded in the JWT and then forwarded. On the receiving side, if the specialized custom HTTP header is attached to any request, the translator tries to extract the users' ID from the request. If successful, the translator acquires a valid service-account access token that has the proper permissions on Keycloak to perform a token exchange with impersonation. With the token exchange, the translator is able to create and fetch a valid access token on behalf of the user. The new access token is attached to the HTTP request and forwarded to the destination. As such, the destination can validate the access token and will receive a valid response from Keycloak. This concept can be adapted to other authentication schemes (e.g. LDAP).

4.6 Automate the Authentication Mesh

The basic concept of the distributed authentication mesh allows the usage of the mesh on all possible platforms. Any platform that wants to participate in the mesh must be able to intercept incoming and outgoing traffic and modify HTTP headers [1]. However, in cloud environments such as Kubernetes, software can be added and removed based on manifest files. In Section 2, the concept of an Operator shows how the Kubernetes API can be extended to manage complex applications. But an Operator is not bound to "manage applications". During the implementation phase of this project, an Operator was created for the distributed authentication mesh. It allows users of Kubernetes to dynamically add and remove applications to the mesh via Custom Resource Definitions (CRDs).

The open-source code of the Operator is hosted on GitHub in the repository https://github.com/WirePact/k8s-operator. The Operator is written in C# with the help of the operator SDK "KubeOps"[32]. "KubeOps" is an SDK that helps with developing Kubernetes Operators. It abstracts certain aspects of Operators, such as the "watcher" logic that needs to be registered within Kubernetes to receive events about certain entities.

[31] Explained in the documentation of Keycloak: https://www.keycloak.org/docs/latest/securing_apps/#_token-exchange

[32] https://github.com/buehler/dotnet-operator-sdk

4.6.1 Use-Cases for the Operator

The Operator must support certain use-cases to have value in the system. It shall help developers to attach applications to the Distributed Authentication Mesh in a declarative way.

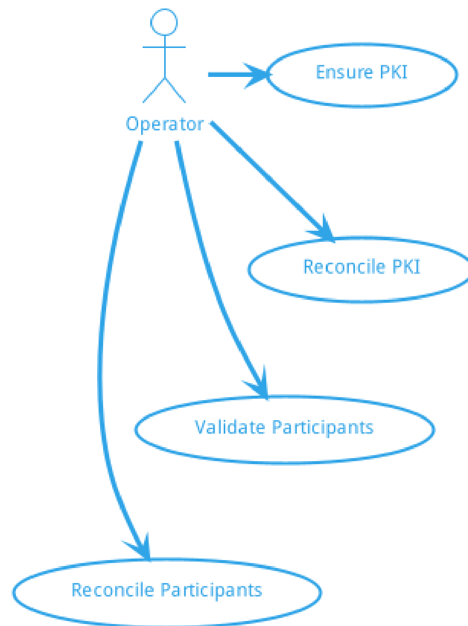

Figure 23: Use-Case Diagram for the Operator

Figure 23 shows the four primary use-cases of the Operator. They are described in a brief form below.

Ensure PKI: The Operator must ensure that there exists a PKI. There must only exist one in the system, otherwise, some participants and translators would use the wrong certificate authority. This results in the inability to communicate with other participants.

Reconcile PKI: The Operator is responsible to create a valid and correctly configured "Deployment", as well as a "Service" for the created PKI. The deployment will run the PKI with the configured container image and the service will allow participants and translators to call the PKI via a system-wide DNS address.

Validate Participants: When a user tries to create a "mesh participant", the Operator is responsible to check if it is valid. For example, the Operator needs to validate that there exists a deployment target and a service that actually want to participate in the

authentication mesh. If one of those vital elements do not exist, the Operator shall reject the participant definition.

Reconcile Participants: The Operator reconciles "mesh participants". Thus, when a participant is created, the Operator modifies the target deployment and injects the required sidecars and modifies service ports to route the communication through the injected proxy application. Furthermore, the Operator must configure the proxy correctly.

4.6.2 Custom Entities for Kubernetes

The Operator uses CRDs to manage and reconcile the participants of the Distributed Authentication Mesh. This section describes the custom entities and their specifications in detail.

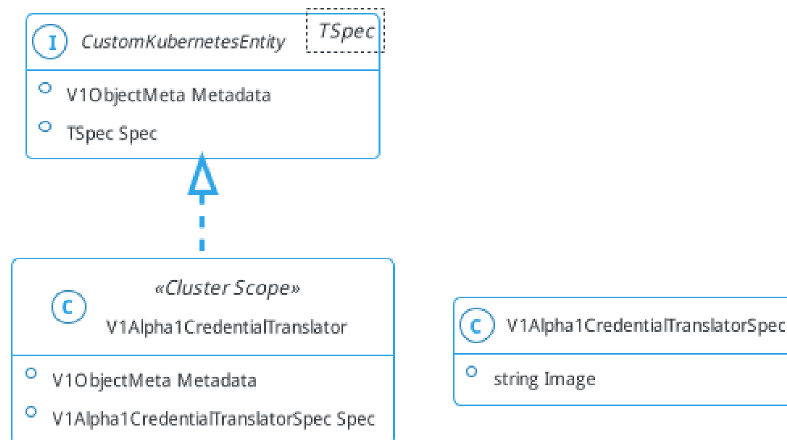

Figure 24: Custom Resource Definition for a Credential Translator

4.6.2.1 Credential Translator The credential translator, as shown in Figure 24, is one of the core elements in the authentication mesh and the automation engine. This CRD defines translators that the Operator and a mesh participant may use. These definitions can be seen as the "inventory" of the Operator that contains the effective container images for translators. This enables developers to create custom translators and inject them into the mesh even if the core system does not support the particular authentication translator.

```yaml
apiVersion: wirepact.ch/v1alpha1
kind: CredentialTranslator
metadata:
  name: basic-auth
```

```yaml
spec:
  image: ghcr.io/wirepact/k8s-basic-auth-translator:latest
```

The declaration above shows an example of such a translator. This is the entity that is stored within Kubernetes when the operator is installed. It does enable the Operator to use the HTTP Basic translator mentioned above. Additionally, the Keycloak OIDC translator is available as well.

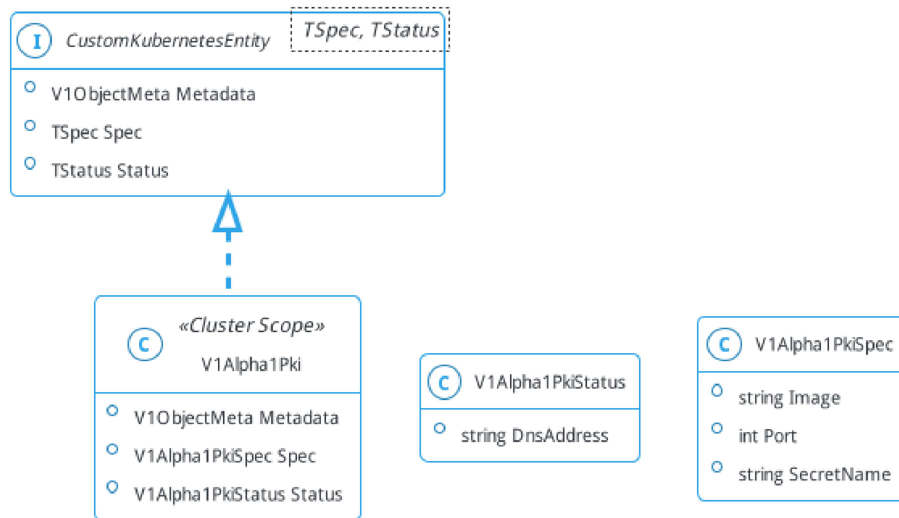

Figure 25: Custom Resource Definition for a PKI

4.6.2.2 PKI Figure 25 shows the definition for a PKI. When the operator fires up, it checks if a PKI already exists. If not, the operator shall create a PKI such that at most one PKI exists for the mesh. The specification contains the container image, a port, and a (Kubernetes-)secret-name. The port defines on which port the PKI will be available for `/ca` and `/csr` calls and the secret name is a reference to a Kubernetes secret. The secret is used to store the serial number, ca certificate, and private key for the PKI. The status of the entity shall be updated by the Operator when a PKI is deployed to the cluster. It must contain the DNS address on which the PKI will be reachable.

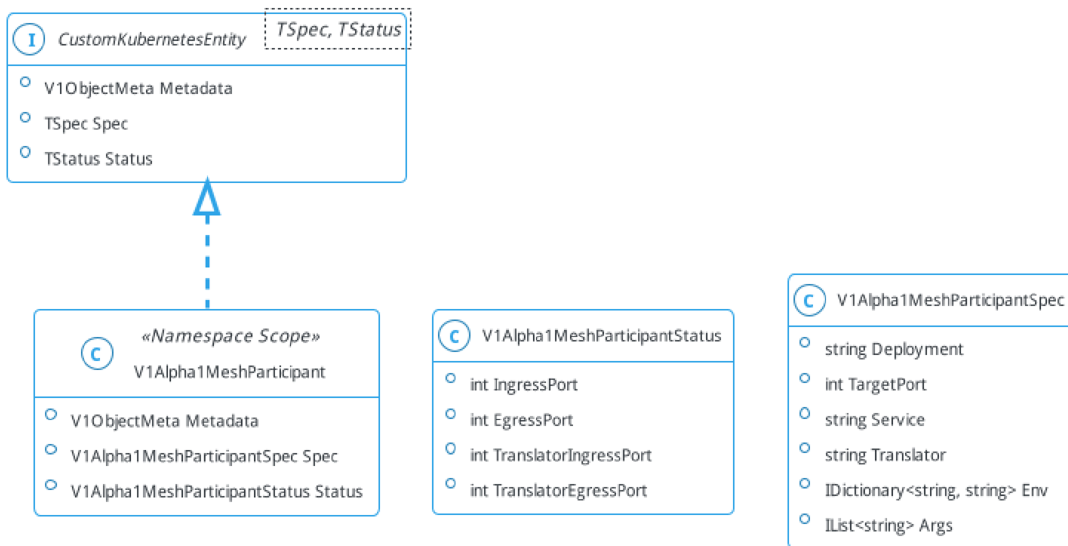

Figure 26: Custom Resource Definition for a Mesh Participant

4.6.2.3 Mesh Participant The mesh participant in Figure 26 enables developers to actually participate in the distributed authentication mesh by defining a deployment and a service. The specification contains the reference to the targeted Kubernetes deployment as well as the Kubernetes service. The target port enables the Operator to correctly configure the Envoy proxy and adjusting the service. The two properties for `Env` and `Args` enable the Operator to set up the translator that is injected as a sidecar into the deployment. It may contain additional environment variables and/or command-line arguments that are attached to the translator. This could be used to configure a translator that needs special information about a user repository or something similar.

```yaml
apiVersion: wirepact.ch/v1alpha1
kind: MeshParticipant
metadata:
  name: participant
spec:
  deployment: deploy
  service: svc
  targetPort: 8080
  translator: keycloak-oidc-translator
  env:
    ISSUER: http://keycloak.localhost/
    CLIENT_ID: demo
    CLIENT_SECRET: very_secret
```

The example participant above shows the specification needed to run the Keycloak OIDC

translator for a deployment ("`deploy`") with a specific service ("`svc`"). The communication on port "`8080`" shall be intercepted by the mesh and the translator in question is "`keycloak-oidc-translator`", for which a `CredentialTranslator` definition with that exact name must exist. As additional environment variables, the issuer, client ID, and client secret variables are passed to the translator such that it can obtain the access tokens from Keycloak.

4.6.3 Managing the Public Key Infrastructure

One of the use-cases mentioned in Figure 23 is managing and reconciling a centralized PKI for the authentication mesh.

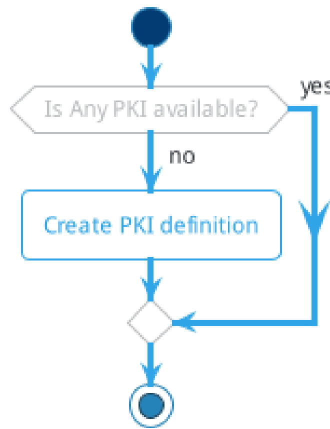

Figure 27: Task for the Startup of the Operator to Ensure a PKI

The startup task in Figure 27 is fairly simple. When the Operator starts, a hosted background service starts (`IHostedService` implementation in .NET) and checks if there are any PKI available. If there are, nothing happens. If not, the Operator creates a PKI entity and stores it within Kubernetes. This will trigger a "normal reconciliation" loop for the entity. This startup process can be further improved by checking the PKI count constantly with a timer instead of just checking when the operator starts. Additionally, a Kubernetes label could ensure that exactly one PKI exists for this particular authentication mesh.

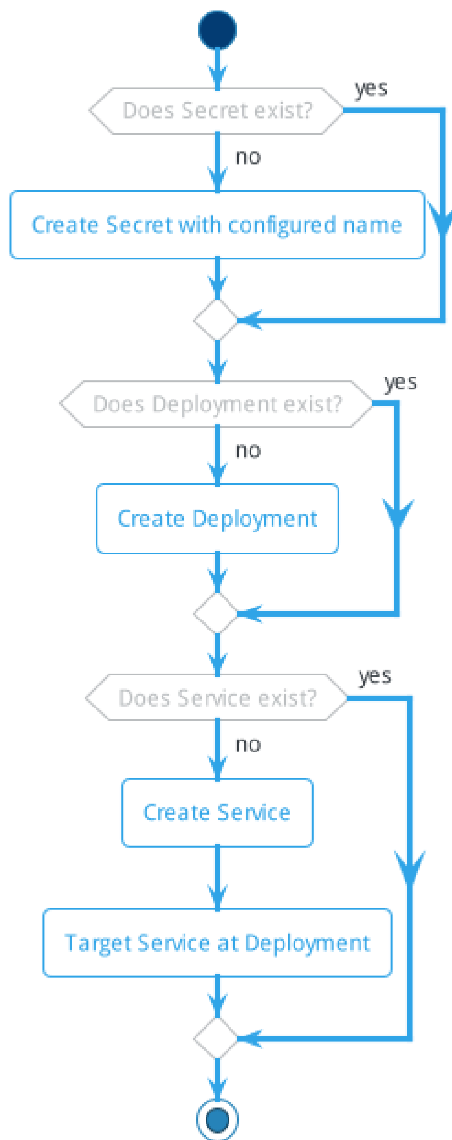

Figure 28: Reconciliation of a PKI

When a reconciliation loop fires for a PKI definition, the Operator takes several steps to deploy the PKI within its own namespace. Figure 28 shows the steps that are needed to reconcile a PKI. First, the Operator checks if the secret (defined in "`secretName`") exists, then if the deployment and the service exist. If any of those elements does not exist, the Operator creates the entities and stores them within Kubernetes. This reconciliation loop is not very complex and only creates a valid deployment as well as a service to enable access to the PKI within Kubernetes.

4.6.4 Reconciling Authentication Mesh Participants

The process to reconcile a mesh participant is more complex than the reconciliation of a PKI. To fulfil the use-case "Reconcile Participants" in Figure 23, the operator needs to adjust the specified deployment and service very heavily. When a reconciliation for a `MeshParticipant` is requested, the Operator performs the following actions one by one:

1. Check if the specified translator (`CredentialTranslator` entity) exists in Kubernetes; if not, throw an error.
2. Check if the specified deployment (only `V1Deployment` at the time of writing, without `StatefulSet` and other deployment types) exists in the namespace of the participant; if not, throw an error.
3. Check and update (if needed) the referenced deployment such that it contains the translator and the envoy proxy as a sidecar.
4. Check and update (if needed) the referenced service such that the correct port of the proxy is used instead of the original one.

It is good practice[33] that an Operator reconciliation loop does not differentiate if an entity was just created or updated within Kubernetes. The reconciliation loop shall check if the desired state still matches the actual state in the system. As such, if the entity gets updated or the Operator is requested to reconcile a participant again by any means, the current objects are checked if they are still valid. As a result, the reconciliation of mesh participants is a complex process. The following two sections show a breakdown of the reconciliation loop for mesh participants.

[33]According to Kubernetes and several operator SDKs like "KubeOps"

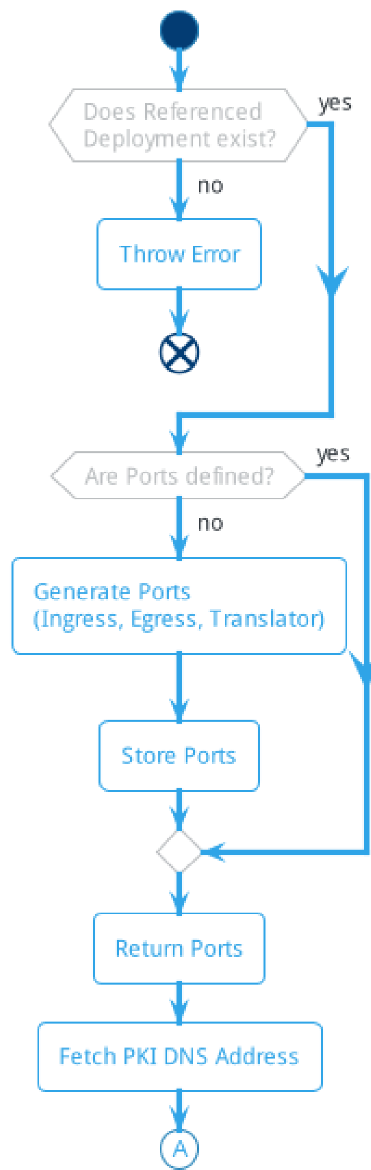

Figure 29: Reconcile Mesh Participant - Deployment - Preparation

4.6.4.1 Reconcile the Target Deployment When the Operator is required to reconcile a mesh participant, several preparation steps are taken. Figure 29 shows the first few actions that reconcile the targeted deployment of a participant. If the referenced deployment does not exist in the given namespace, an error is thrown. Further, if the participant does not contain defined ports (for incoming, outgoing, transformer-incoming, and transformer-outgoing communication), they are created and stored. Otherwise, the ports are returned. The last step in preparation is fetching the DNS address for the

PKI.

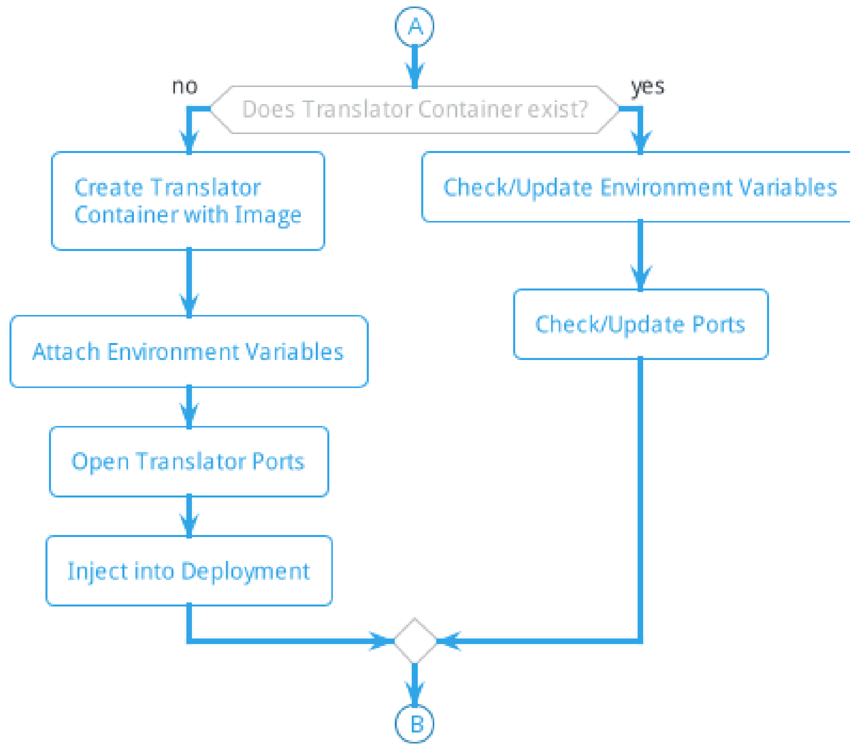

Figure 30: Reconcile Mesh Participant - Deployment - Translator Container

When the Operator finishes the preparation in Figure 29, the process in Figure 30 takes place. The Kubernetes deployment is analyzed more thoroughly. The first check targets the sidecar for the credential translator. If the container definition does not exist, it is created and the environment variables and ports are configured. Then it is attached to the deployment. If the container did exist, it is validated and checked that all required values are as they should be. This step mitigates the risk that the manifest are edited externally since the Operator will reset any changes to the sidecars.

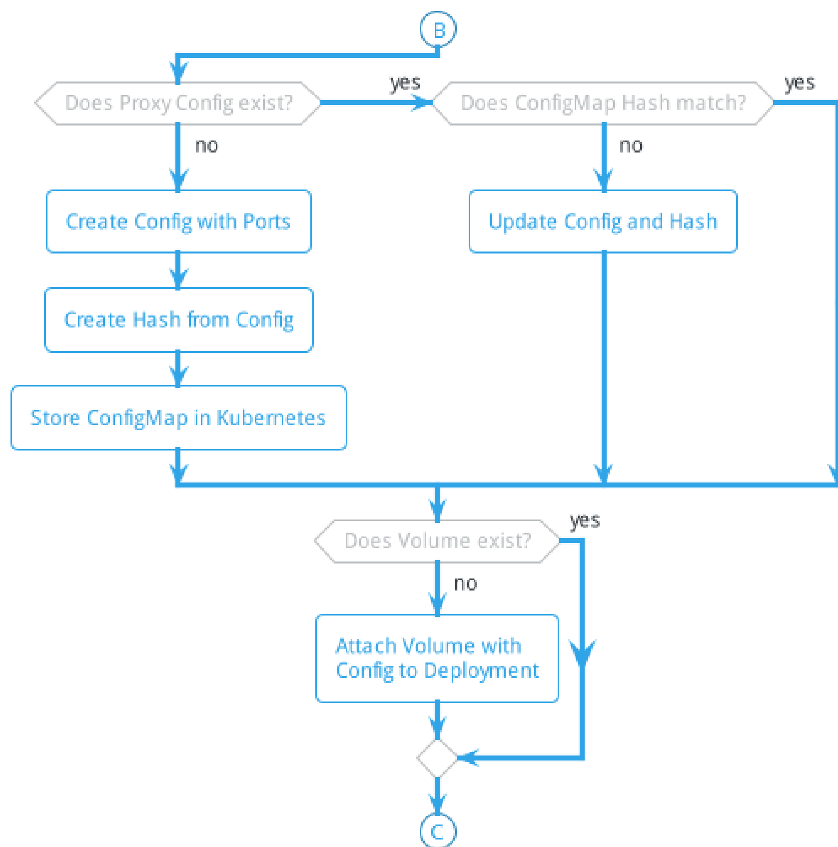

Figure 31: Reconcile Mesh Participant - Deployment - Envoy Configuration

The next step, as Figure 31 depicts, is validating the proxy (Envoy) configuration. The config is stored in a `V1ConfigMap` in Kubernetes. The config object contains two properties ("`envoy-config.yaml`" and "`config-hash`") that contain the config and a SHA256 hash of the config. If the config object does not exist, the config is created and stored in the config object with the hash. If it does exist, the hash within the object is checked against the config that **should** be stored. When the hash matches, nothing happens. Otherwise, the generated config is stored along with its hash. After the config is validated, the deployment is searched if the associated volume to bind the config as files exists. If not, it is attached to the deployment.

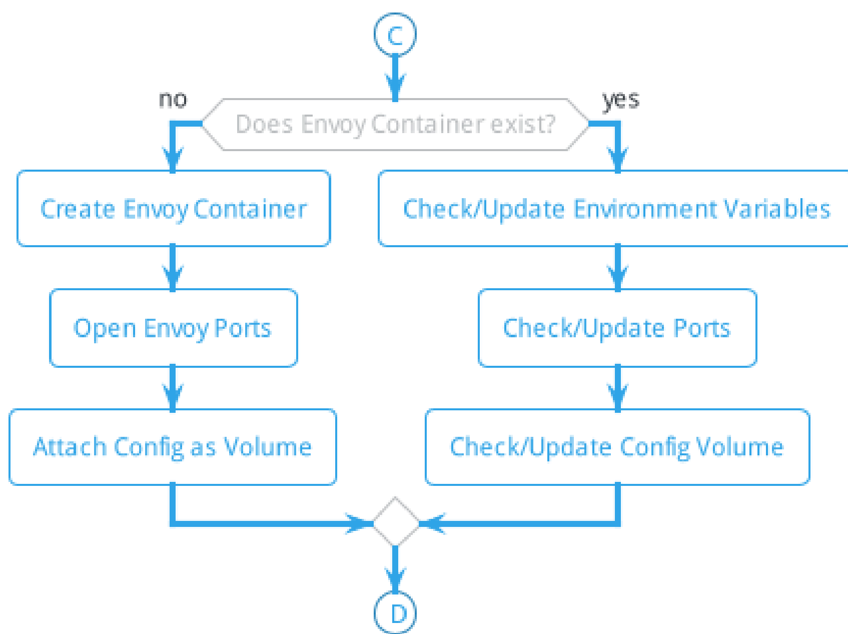

Figure 32: Reconcile Mesh Participant - Deployment - Envoy Container

Second to last step is checking the Envoy container. Figure 32 shows these steps. The same technique that checks the translator container in Figure 30 ensures the existence and correctness of the Envoy container. The container contains several environment variables and open ports. Further, the container is injected into the deployment when it does not exist.

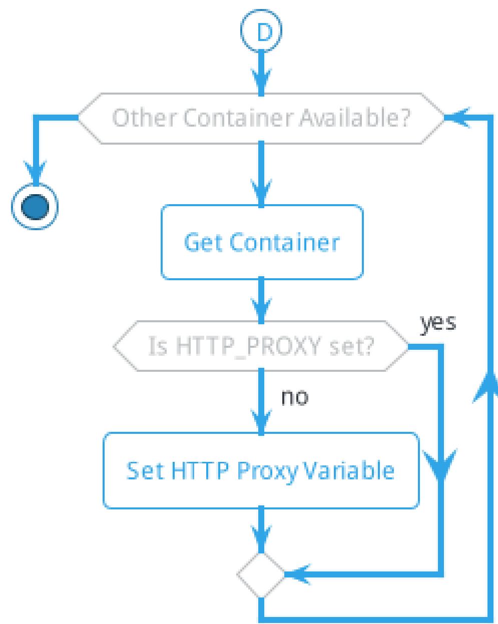

Figure 33: Reconcile Mesh Participant - Deployment - Proxy Environment Variable

Figure 33 shows the last step to reconcile the targeted deployment for a mesh participant. All other containers in the deployment receive an environment variable with the local HTTP proxy. The variable `HTTP_PROXY` contains the value "`http://localhost:{egressPort}`". Since multiple containers in a pod run on the same "machine", they share the same localhost. This enables the Distributed Authentication Mesh to configure the application to use a local running Envoy instance as HTTP proxy. Hence, the application routes its outgoing communication through Envoy which then in turn can communicate with the credential translator.

4.6.4.2 Reconcile the Target Service In contrast to the target deployment reconciliation explained in the section above, the process to reconcile the targeted service is not as complex.

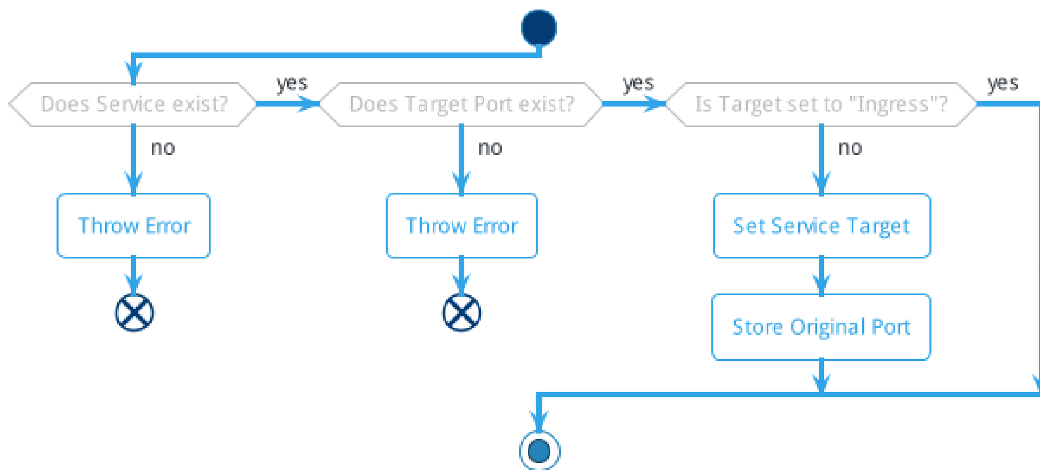

Figure 34: Reconcile Mesh Participant - Service

Figure 34 shows the steps to reconcile a target service for a mesh participant. If the service does not exist, or it does not contain the target port that is specified in the participant entity, an error is thrown. The target port for this "external service port" is then changed to "`ingress`". The referenced deployment contains this "ingress" reference as external application port for incoming communication on the Envoy sidecar. As such, all incoming communication on the target port is routed to Envoy. Thus, Envoy can intercept the communication and can consult with the credential translator.

5 Conclusions and Outlook

This project further improved the core concept of the “Distributed Authentication Mesh” proposed in “Distributed Authentication Mesh” [1]. The goals of this project were the definition and implementation of the “common language format” and a complete implementation of in a cloud environment (i.e. “Kubernetes”).

Section 1 introduces the reader into the general topic of the project and shows references to past work. Further, the past work is briefly analyzed and the goals for this project are outlined.

Section 2 defines the scope of this project and introduces readers into the technologies used by this project. Kubernetes, the central technology in this project, and some of the required patterns, such as the Operator pattern and the Sidecar pattern, are explained. Section 2 also gives an overview of the used security mechanisms in this project.

The next section, Section 3, describes the actual state of the “Distributed Authentication Mesh”. The concept of the mesh only exists in a theoretical form and the only form of proof that exists is a Proof of Concept (PoC). The PoC does show that it is possible to change HTTP headers for HTTP requests in-flight but it does not show how this may be used for the mesh. Also, the often referenced “common language format” is not defined nor implemented.

The core of this project, the analyzation and implementation of the common language and a practical implementation of the mesh, resides in Section 4. Since the concept of the “Distributed Authentication Mesh” only uses the term “common language format”, a useful implementation of the system was not possible.

Section 4 analyzes various data formats, such as structured formats (JSON, YAML, etc.), x509 certificates, and JSON Web Tokens (JWT). Then, a comparison of the formats shows the decision process for the JWT format. Structured formats would be feasible for such a use-case, but they lack the possibility of validating the integrity of the data without implementing further concepts. On the other hand, x509 certificates provide such a mechanism and are already enabled though the existence of a Public Key Infrastructure (PKI) in the authentication mesh. The storage of custom data is possible withing certificates [13, Sec. 4]. However, they tend to be cumbersome to use in various programming languages. Since one goal of the authentication mesh is a good developer experience, x509 are not the most likely choice. JSON Web Tokens are selected as common language format because of the ease-of-use and the possibility to sign them with an algorithm [14]. The JSON Web Signature (JWS) [15] in conjunction with a JWT enables data to be transmitted securely.

The reminder of Section 4 shows the definition and implementation of the PKI, a translator base, an HTTP Basic translator, an OIDC translator, and the Kubernetes Operator that automates the whole authentication mesh. All these sub-sections show (where needed) use-case diagrams and process/sequence/invocation diagrams to explain the

implementation further. All created software is available as open-source on GitHub under the organization “WirePact” (https://github.com/WirePact). To test the authentication mesh in Kubernetes, the GitHub organization provides a demo application that installs the Operator and a simple application with “Keycloak” and Basic Auth. The code and installation instruction can be found on GitHub (https://github.com/WirePact/k8s-demo-application).

The goal for future work is to specify and implement the concept of a “Rule Engine” to further improve the “Distributed Authentication Mesh”. To provide additional use for the mesh, a rule based access engine could enhance the usefulness of the distributed authentication mesh. Such a rule engine would further improve the security of the overall system. This engine takes the configuration from the configuration store and takes place before the translator checks the transmitted identity. A partial list of features could be:

- Timed access: define times when the access to the service is rejected or explicitly allowed.
- IP range: Define IP ranges that are allowed or blocked. This could prevent cross-datacenter-access.
- Custom logic: With the power of a small scripting language[34], custom logic could be built to allow or reject access to services.

The rule engine should be extensible such that additional mechanisms can be included into it. There are other useful filters that help development teams all over the world to create more secure software.

In this project, the state-of-the-art of the concept and the implementation base on the assumption that all participants reside in the same trust context. Extending the concepts and the implementation of this project to enable the “Distributed Authentication Mesh” to be truly “distributed”.

With the implementation of the authentication mesh, as documented in Section 4, the system can be used in a Kubernetes cloud environment. This enables developers and companies to use legacy applications in conjunction with modern state-of-the-art software without changing one of the mentioned applications. The authentication mesh does dynamically transform user information from the source system into a signed JWT and back to the authentication scheme of the destination.

[34]For example Lua or JavaScript with their respective execution environment

Bibliography

[1] C. Bühler, “Distributed authentication mesh,” University of Applied Science of Eastern Switzerland (OST), 2021. Available: https://buehler.github.io/mse-project-thesis-1/report.pdf

[2] J. Humble and D. Farley, *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[3] B. Burns, J. Beda, and K. Hightower, *Kubernetes*. Dpunkt, 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, Inc., 2020.

[6] prometheus-operator, “Prometheus operator docs.” Available: https://prometheus-operator.dev/docs/

[7] 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

[8] J. Reschke, “The ’Basic’ HTTP authentication scheme,” Internet Engineering Task Force IETF, RFC, 2015. Available: https://tools.ietf.org/html/rfc7617

[9] D. Hardt *et al.*, “The OAuth 2.0 authorization framework,” Internet Engineering Task Force IETF, RFC, 2012. Available: https://tools.ietf.org/html/rfc6749

[10] 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

[11] 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.

[12] T. Santos and C. Serrão, “Secure javascript object notation (SecJSON) enabling granular confidentiality and integrity of JSON documents,” in *2016 11th international conference for internet technology and secured transactions (ICITST)*, 2016, pp. 329–334. doi: 10.1109/ICITST.2016.7856724.

[13] D. Cooper, S. Boeyen, S. Santesson, T. Polk, R. Housley, and S. Farrell, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” Internet Engineering Task Force IETF, RFC 5280, May 2008. doi: 10.17487/RFC5280.

[14] M. B. Jones, Bradley John, and N. Sakimura, “JSON web token (JWT),” Internet Engineering Task Force IETF, RFC, May 2015. Available: https://tools.ietf.org/html/rfc7519

[15] M. B. Jones, Bradley John, and N. Sakimura, "JSON web signature (JWS)," Internet Engineering Task Force IETF, RFC, May 2015. Available: https://tools.ietf.org/html/rfc7515