Title*

Some SubTitle

Christoph Bühler

Autumn Semester 2021

University of Applied Science of Eastern Switzerland (OST)

This is the abstract. TODO.

*I'd like to say thank you :)

Contents

List of Tables

List of Figures

Declaration of Authorship

I, Christoph Bühler, declare that this project report titled, “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, December 5, 2021

Christoph Bühler

1 Introduction

With the introduction of the concept “Distributed Authentication Mesh” [1], a theoretical base for dynamic authorization was created. The project, in conjunction with the Proof of Concept (PoC) showed, that it is generally possible to transform the identity of a user to another application. This transmitted identity can authenticate the user in a trusted system. In contrast to SAML (Security Assertion Markup Language) the authentication mesh does not require all parties to understand the same authentication and authorization mechanism. The mesh is designed to work within a heterogeneous authentication landscape.

This project enhances the concept of the “Distributed Authentication Mesh” by evaluating and specifying the transport protocol for the common language between services. The common language is crucial for the success of a

The remainder of the report describes prerequisites, used technologies and further concepts.

2 Definitions and Clarification of the Scope

This section provides general information about the project, the context, and prerequisite knowledge. It gives an overview of the context as well as terminology and general definitions.

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" that is mentioned in the former project. This project provides an analysis of various methods to specify and implement such a common language and gives an implementation for the selected common format. Additionally, it extends the authentication mesh with a rule engine that allows conditional access control to the destinations of the mesh. As an example, one could configure specific IP ranges that are blocked, specific times when access is allowed or completely free logic with the help of a scripting language[1].

in scope:

- pki
- signed JWT
- communication

out of scope:

- service mesh (service discovery)
- auth PKI
- cert revoke etc.

2.2 Kubernetes and its Patterns

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

[1] Such as Lua: https://www.lua.org/

2.2.1 Kubernetes, the Orchestrator

Kubernetes is an orchestration software for containerized applications. Originally developed by Google and now supported by the Cloud Native Computing Foundation (CNCF) [2, Ch. 1]. Kubernetes manages the containerized applications and provides access to applications via “Services” that use a DNS naming system. Applications are described in a declarative way in either YAML or JSON.

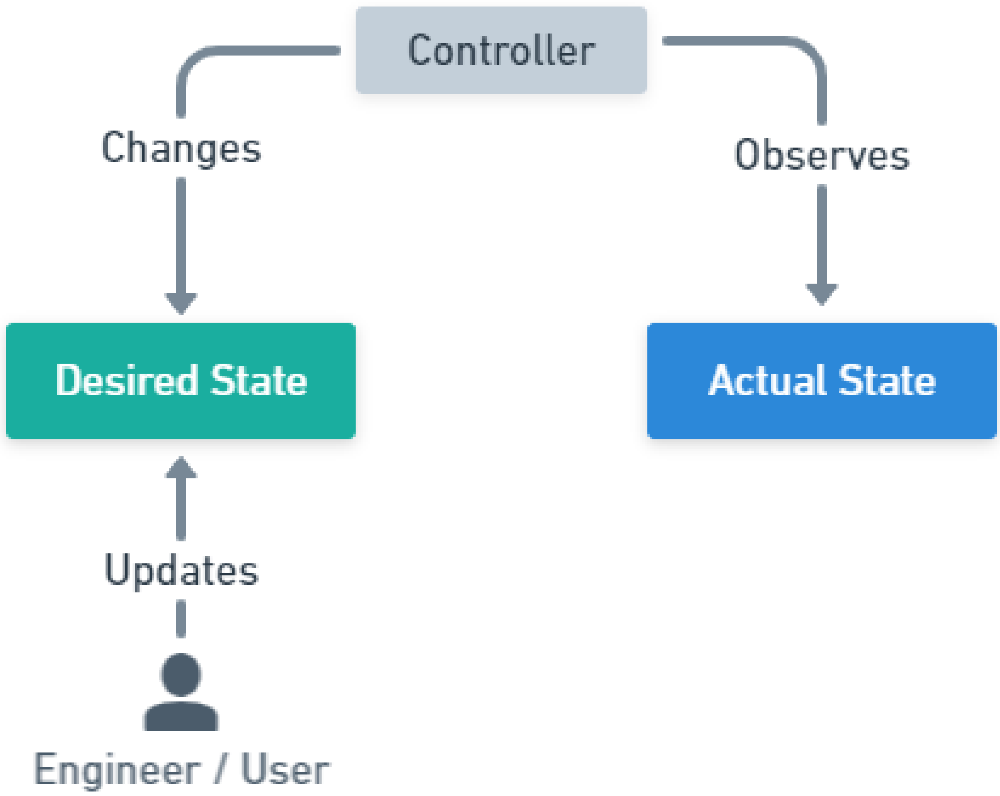

Figure 1: The Kubernetes Control Loop

Figure 1 shows the Kubernetes “control loop.” While it is not a “loop” in the common sense, the technique behind the system is a 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 take action and starts another instance such that the actual state matches the desired state.

2.2.2 An Operator, the Reliability Engineer

The API of Kubernetes is extensible with custom API endpoints (so-called custom resources). With the help of "CustomResourceDefinitions" (CRD), a user can extend the core API of Kubernetes with their own resources [2, Ch. 16]. With the help of these extensions, an Operator can watch the custom resources and act as a controller for certain resources.

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 [3].

Two example operators:

- Prometheus Operator[2]: Manages instances of Prometheus (open-source monitoring and alerting software).
- Postgres Operator[3]: Manages PostgreSQL clusters in Kubernetes.

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

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[4]: GoLang[5] Operator Framework
- KubeOps[6]: .NET Operator SDK
- Operator SDK[7]: SDK that supports GoLang, Ansible[8] or Helm[9]
- shell-operator[10]: Operator that supports bash scripts as hooks for reconciling

Operators are the most complex extension possibility for Kubernetes, but also the most powerful one [2]. With Operators, whole applications can be automated in a declarative and self-healing way.

[2] https://github.com/prometheus-operator/prometheus-operator
[3] https://github.com/zalando/postgres-operator
[4] https://book.kubebuilder.io/
[5] https://golang.org/
[6] https://buehler.github.io/dotnet-operator-sdk/
[7] https://operatorframework.io/
[8] https://www.ansible.com/
[9] https://helm.sh/
[10] https://github.com/flant/shell-operator

2.2.3 A Sidecar, the Extension

Sidecars enhance a "Pod"[11] by injecting additional containers to the defined one [4]. As an example: A containerized application runs in its Docker[12] image and writes logs to `/var/logs/app.log`. A specialized "log-reader" 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[13].

2.3 Securing Communication

This section provides the required knowledge about security for this report. Authentication and authorization are big topics in software engineering and there are various standards 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.

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. 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` [5].

2.3.2 OpenID Connect

OIDC (OpenID Connect) is defined 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 [6].

OIDC extends OAuth with authentication, 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 an identity [7]. As an example, a user wants to access a protected API via web GUI. The user is forwarded to an external login page and enters the credentials. When they

[11] The smallest possible workload unit in Kubernetes. A Pod contains of one or more containers that run a containerized application image.

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

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

are correct, the user gets redirected to the web application which can fetch an access token for the user. This access token can be transmitted to the API to authenticate and authorize the user. The API is able to verify the token with the login server and can reject or allow the request.

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.

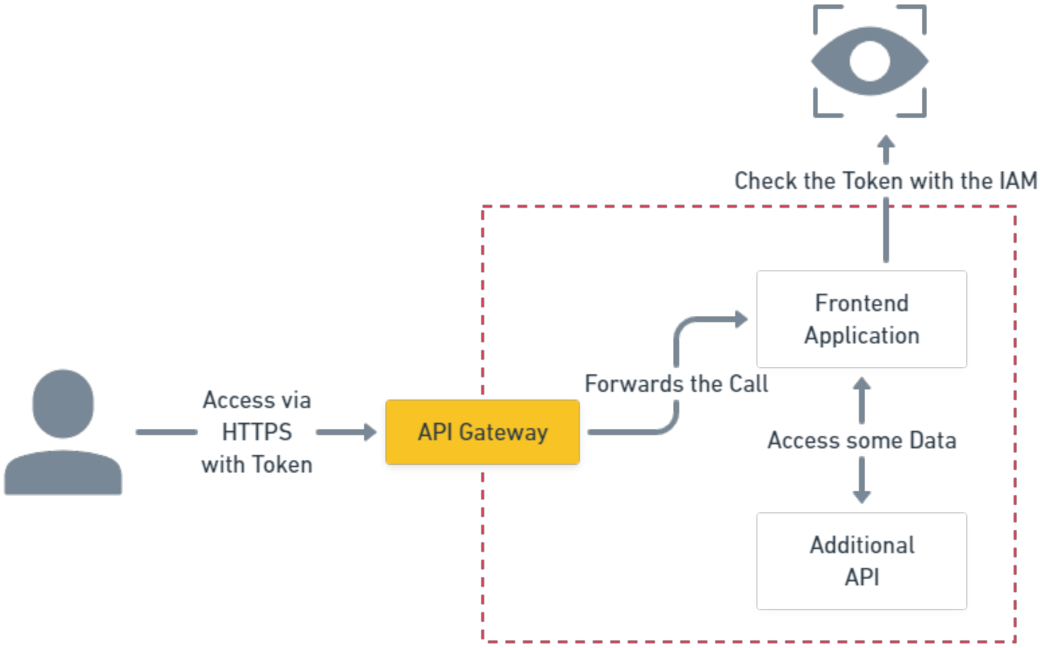

Figure 2: Example of a Trust Zone

As an example, we consider Figure 2. The API gateway is the, hopefully, 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 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 [8]. Zero trust assumes that every call could be an intercepted by an attacker. Therefore, all requests must be validated. As a consequence, the frontend in Figure 2 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.

3 State of the Authentication Mesh and the Deficiencies

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.

3.1 Common Language Format for Communication

The "Distributed Authentication Mesh" defines an architecture that enables a dynamic conversion of user identities in a declarative way [1]. The common language format however, is neither defined nor implemented. To enable the possibility of a production-grade software based on the concepts of the authentication mesh, this part must be specified.

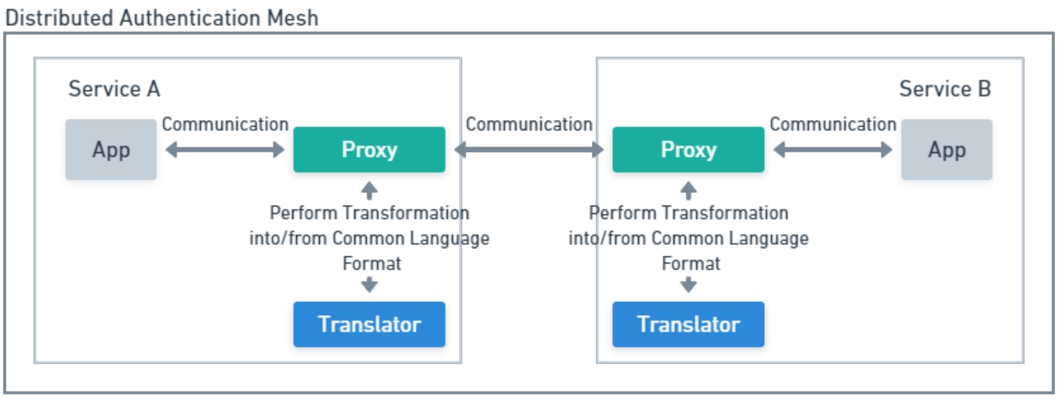

Figure 3: General communication flow of two services in the distributed authentication mesh

Figure 3 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.

This 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.

3.2 Restricting Access to Services with Rules

In the concept of the authentication mesh, a proxy intercepts the communication from and to applications that are part of the mesh. The interception does not interfere with the data stream. The goal of the intrusion is the de-, and encoding of the user identity that is transmitted [1]. To provide additional use for this system, a rule based access engine could enhance the usefulness of the distributed authentication mesh.

An additional mechanism ("Rule Engine") could be added to the mesh. This engine takes the configuration from the 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[14], 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.

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

4 Implementing a Common Language and Conditional Access

This section analyzes different approaches to create a common language format between the service 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.

4.1 Goals and Non-Goals of the Project

This section provides two tables with functional and non-functional requirements for the project. The mentioned tables extend the requirements of the distributed authentication mesh [1], seq. 4.2. 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.

tbl.

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. 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 is made.

4.2.1 YAML, XML, JSON, and Others

YAML (YAML Ain't Markup Language[15]), XML (Extensible Markup Language[16]), JSON (JavaScript Object Notation[17]) 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
```

[15] https://yaml.org/
[16] https://www.w3.org/XML/
[17] https://www.json.org/

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 (TODO REF: https://ieeexplore.ieee.org/abstract/document/7856724) approaches like "SecJSON" that enhance JSON with security and integrity mechanisms. But if the standard of the specifications is used, no integrity check can be perfromed and the translators of the authentication mesh cannot validate if the data was not modified. Thus, using a "simple" structured dataformat for the transmission of the user identity would not suffice the security requirements of the system.

TODO: Search other secure methods for xml/yaml TODO: describe a possible way to implement integrity hashing?

4.2.2 X509 Certificates

The x509 standard defines how certificates shall be used. Today, the connection over HTTPS is done via TLS and certificate encryption. (TODO CHECK!) The data in a certificate is not fixed, however. There can be "private extensions" (TODO REF to paper/spec) that can be used to transmit data to a possible receiver.

Certificates have the big advantage that they can be integrity checked via already implemented hashing mechanisms and provide a "trust anchor"[18] in the form of a root certificate authority (root CA).

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[19] on the other hand, has a much better support for manipulating x509 certificates. Since the result of this project should have a good developer experience, using x509 certificates is not be the best solution to solve the communication and integrity issue.

TODO: example of a certificate TODO: example of private fields TODO: ref for private x509 fields

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 (TODO REF). A JWT contains three parts (TODO REF): A "header," a "payload" and the "signature." 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

[18] A trust anchor is a root for all trust in the system.
[19] https://go.dev/

transmitted. The last part of the JWT contains the constructred signature of the header and the payload. This signature is constructed by either a symmetrical or asymmetrical hashing algorithm. (TODO REF, HMAC256 or RSA256).

In OpenID Connect (OIDC) (TODO REF), “Bearer” tokens are used to authenticate a user against a system. The tokens may be opaque or transparent (TODO REF).

- x5c / x5t TODO: ref JOSE TODO: ref JWT / JWS / JWK

4.3 Implementing a Secure Common Identity

- implement the PKI
- usage of PKI key material
- use key material to sign JWT tokens
- “translator” can validate JWT tokens with key material

4.4 Intercept the traffic

differentiate between service mesh (service discovery) and HTTPPROXY

5 Conclusions and Outlook

Bibliography

[1] C. Bühler, “Distributed authentication mesh,” 2021.

[2] B. Burns, J. Beda, and K. Hightower, *Kubernetes.* Dpunkt, 2018.

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

[4] 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

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

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

[7] 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

[8] 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.

Appendix A - if any

Some Appendix