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, January 10, 2022

Christoph Bühler

1 Introduction

With the introduction of the concept “Distributed Authentication Mesh” [1], a theoretical base for dynamic authorization in heterogeneous systems 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 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 mechanism. The mesh is designed to work within a heterogeneous authentication landscape.

The PoC was designed to show the ability of heterogeneous authentication. 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. The common language is crucial for the success of the mesh. Additionally, an implementation of the mesh for Kubernetes[1] is created during this project. The implementation is tied to Kubernetes itself, but the concept of the mesh can be adapted to various platforms.

The remainder of the report describes prerequisites, used technologies with their terminology and further concepts. The state of the authentication mesh shows the current version of the concept and which elements are missing. The implementation shows the concrete evaluation of the common language format combined with the definition and implementation 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.

[1]https://kubernetes.io/

2 Definitions and Clarification of the Scope

This section provides general information about the project, the context, and prerequisite knowledge. The scope shows 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" that is mentioned in the former project. This project analyses various patterns[2] for such a common language and further implements the common language in Kubernetes. 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.

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. As an alternative, the mentioned software could be configured with environment variables or command line arguments to receive the information about the proxy location.

Another topic that is not in the scope of this project is the authentication and authorization of services against the PKI. While there exist mechanisms to authenticate services, like the usage of a pre-shared key, it is not part of the scope of this project since all elements 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 in 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 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," 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 entrypoint to the system. The Ingress points to a service and provides centralized routing from the outside world into some application running in Kubernetes. The Ingress may contain definitions for hostnames or path informations 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"[4] and "Ambassador"[5].

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) [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.

[4]https://www.nginx.com/

[5]https://www.getambassador.io/

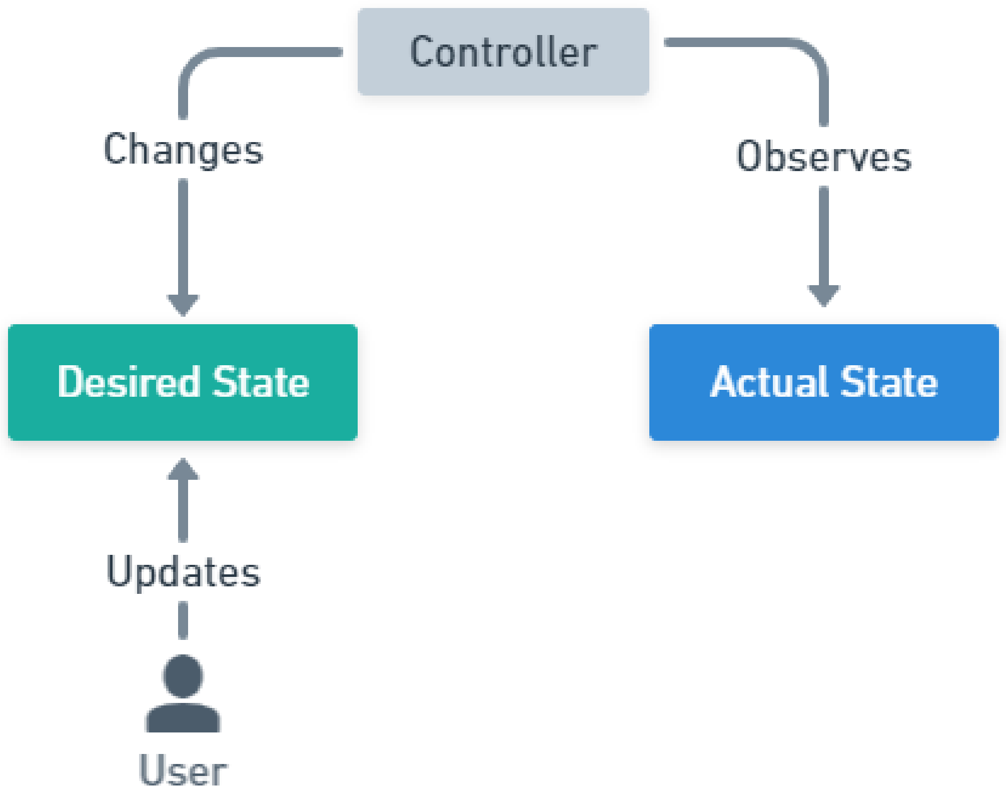


Figure 1: The Kubernetes Control Loop

Figure 1 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 state.

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" (CRD), a user can extend the core API of Kubernetes with their own resources [2, 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 1.

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[6]: Manages instances of Prometheus (open-source monitoring and alerting software).
- Postgres Operator[7]: 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` [4]. 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`[8] that runs the Prometheus Docker image with the scale of two instances. 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[9] [4].

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:

[6]https://github.com/prometheus-operator/prometheus-operator
[7]https://github.com/zalando/postgres-operator
[8]A form of deployment like `Deployment` but with certain stateful mechanics inside Kubernetes.
[9]Scraping: fetch the metrics from the target system and store them with time information

- kubebuilder[10]: Go[11] Operator Framework
- KubeOps[12]: .NET Operator SDK
- Operator SDK[13]: SDK that supports Go, Ansible[14] or Helm[15]
- shell-operator[16]: Operator that supports bash scripts as hooks for reconciling

Operators, and software that implements the Operator pattern, 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.

2.2.4 A Sidecar, the Extension to a Pod

Sidecars enhance a "Pod"[17] by injecting additional containers to the defined one [5].

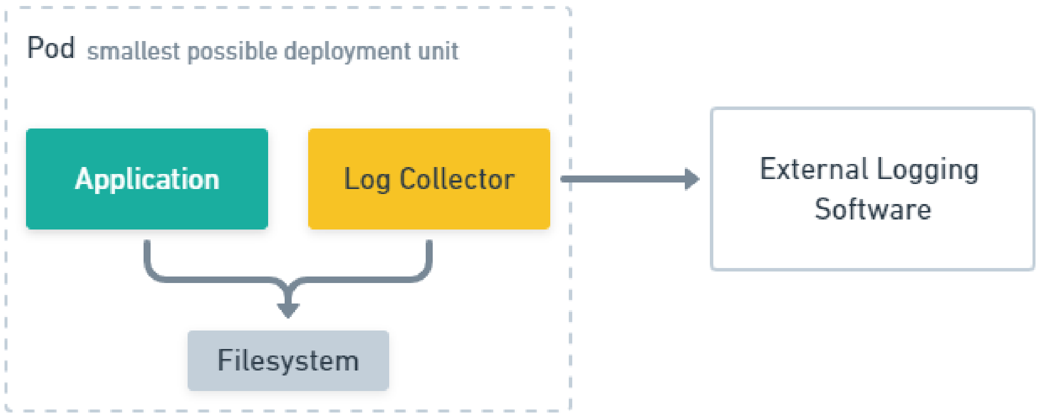

Pod smallest possible deployment unit
Application
Log Collector
Filesystem
External Logging Software

Figure 2: A Sidecar example

Figure 2 shows an example: A containerized application runs in its Docker[18] image and writes logs to `/var/logs/app.log` in the shared file system. A specialized "Log

[10]https://book.kubebuilder.io/
[11]https://golang.org/
[12]https://buehler.github.io/dotnet-operator-sdk/
[13]https://operatorframework.io/
[14]https://www.ansible.com/
[15]https://helm.sh/
[16]https://github.com/flant/shell-operator
[17]The smallest possible workload unit in Kubernetes. A Pod contains of one or more containers that run a containerized application image.
[18]https://www.docker.com/

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[19].

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 [2, 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.

2.3.1 HTTP Basic Authentication

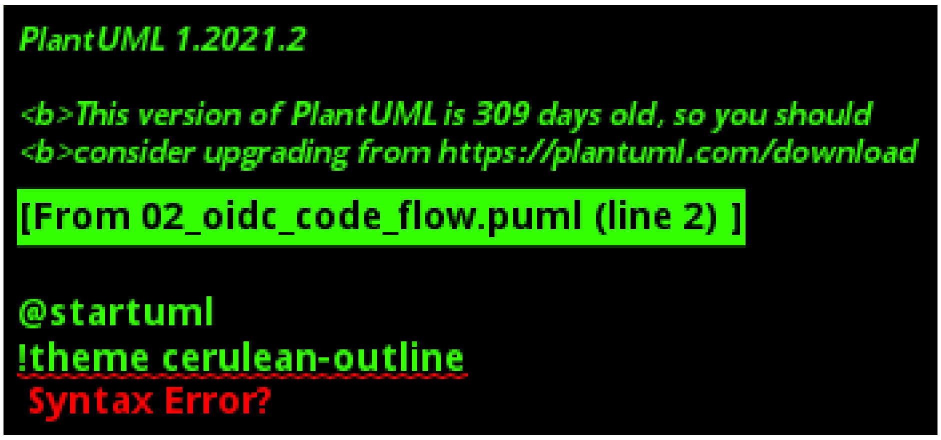
The "Basic" authentication scheme is defined in **RFC7617**. Basic is a trivial authentication scheme which provides an extremely low security when used without HTTPS. It does not use any real form of encryption, nor can any party validate the source of the data. To transmit basic credentials, the username 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` [6]. 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 [7].

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 an identity [8].

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



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 02_oidc_code_flow.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 3: OIDC Authorization Code Flow

As an example, Figure 3 shows a user that wants to access a protected API. The user is forwarded to an external login page (Identity Provider) and enters the credentials. When they are correct, the user gets redirected to the web application with an authorization code. The code is provided to the application (Relying Party) which in turn can fetch an access and ID token for the user. These tokens identify, authenticate and authorize the user on relying systems. A GUI would be able to initiate this process and then forward the fetched tokens to an API. 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.

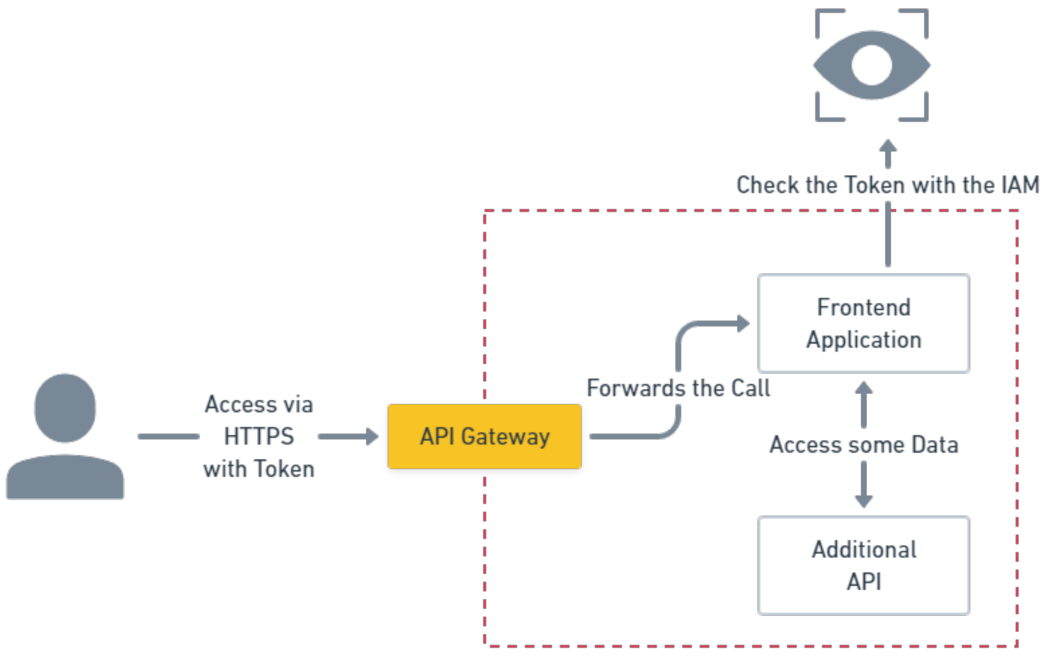


Figure 4: Example of a Trust Zone

As an example, we consider Figure 4. 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 3, 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 [9]. 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 4 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 a perfect safe solution. But a partial solution is to use well-known software[20] and applications to provide the safest possible implementations of such software.

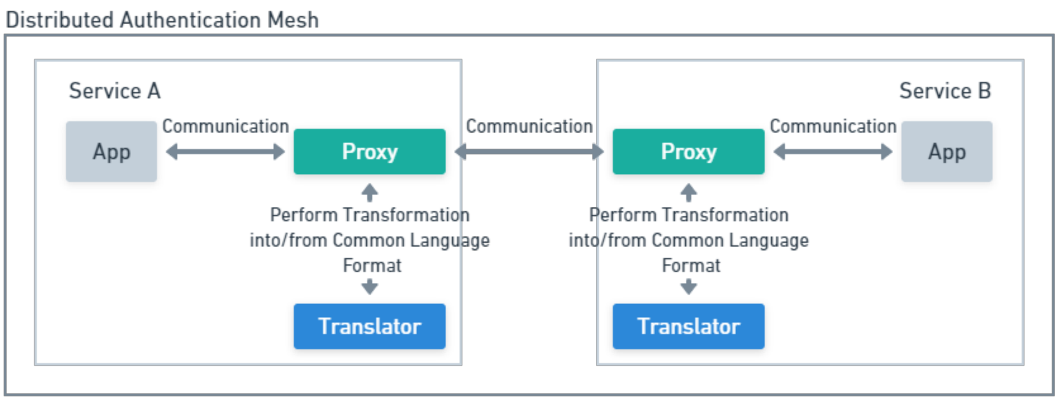
[20]For example “Auth0,” “Keykloak” or “Octa” among others.

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.

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 yet. Past work did implement a Proof of Concept (PoC) to show the general idea, but did not prove the feasibility with the common langauge format. To enable the possibility of a production-grade software based on the concepts of the authentication mesh, the common language must be defined and specified.



Distributed Authentication Mesh
Service A
App
Communication
Proxy
Perform Transformation into/from Common Language Format
Translator
Communication
Service B
Proxy
Perform Transformation into/from Common Language Format
Translator
Communication
App

Figure 5: General communication flow of two services in the distributed authentication mesh

Figure 5 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.

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 proxy 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. Such a rule engine would further improve the security of the overall system.

An additional mechanism ("Rule Engine") could be added to the mesh. 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[21], 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. This rule engine however, is not implemented in this project. This project focuses on the common language used to transmit the user identity as well as the implementation of the authentication mesh in a productive environment. Further work may define and implement the rule engine as an addition to the authentication mesh.

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

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].

Table 1: Functional Requirements

Name	Description
REQ 1	The translator generates/parses the common language format.
REQ 2	The translator handles errors according to the previous requirements.
REQ 3	The translator is authorized to perform its actions.
REQ 4	The common format contains a way to validate the integrity.

Table 1 shows the functional requirements for this project.

Table 2: Non-Functional Requirements

Name	Description
NFR 1	The common language contains all needed information to identify a user.
NFR 2	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.
NFR 3	The common format can be validated with the PKI.

TODO: other table fmt.

Table 2 shows the non-functional requirements for this project.

Table 1 and Table 2 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 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 is made.

4.2.1 YAML, XML, JSON, and Others

YAML (YAML Ain't Markup Language[22]), XML (Extensible Markup Language[23]), JSON (JavaScript Object Notation[24]) 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 [10]. But if the standard of the specifications 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

[22]https://yaml.org/
[23]https://www.w3.org/XML/
[24]https://www.json.org/

fields do 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 [11, Ch. 4]. These extensions could be used to transmit the data needed for the distributed authentication mesh.

Certificates have the big advantage: they can be integrity checked via already implemented hashing mechanisms and provide a “trust anchor”[25] in the form of a root certificate authority (root CA). Furthermore, if certificates would be used to transmit the users' 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[26] 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” [12]. 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

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

digital signature [13]. 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 [13].

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

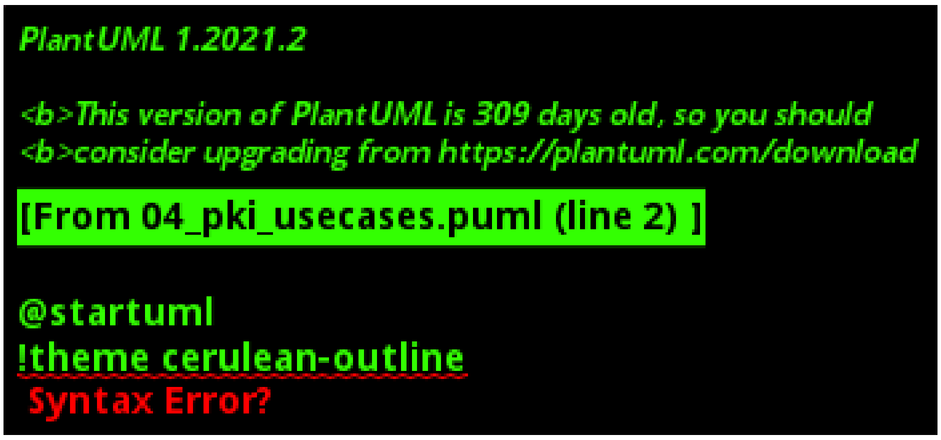
After considering the possible transport formats above, we can now analyze the pro and contra arguments. While **structured formats** like YAML and JSON are widely known and easily implemented, they do not offer a built-in mechanism to prevent data manipulation and integrity checking. There are standards 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 6.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_pki_usecases.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 6: 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

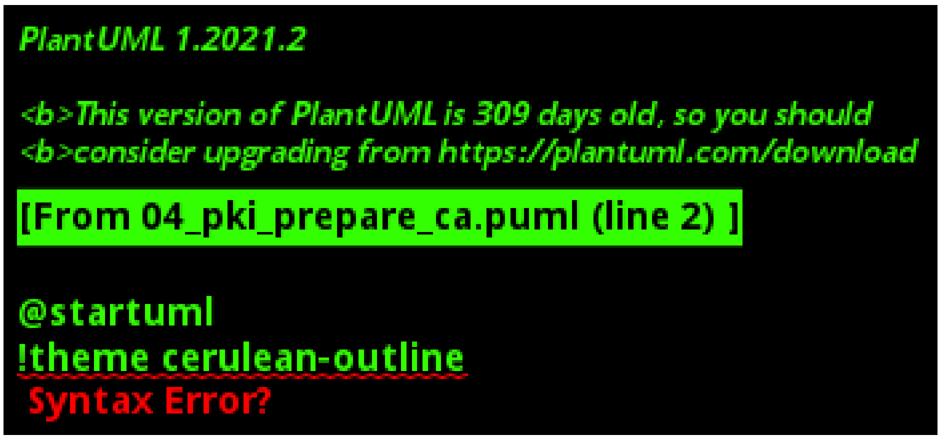
To manage HTTP request to the PKI, the "Gin"[27] framework is used. It allows easy management of routing and provides support for middlewares if needed. To set up the CA, the following code enables a web server with the needed routes to `/ca` and `/csr`:

```go
router := gin.Default()

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

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

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.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_pki_prepare_ca.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 7: 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 7 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 in 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 7 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.

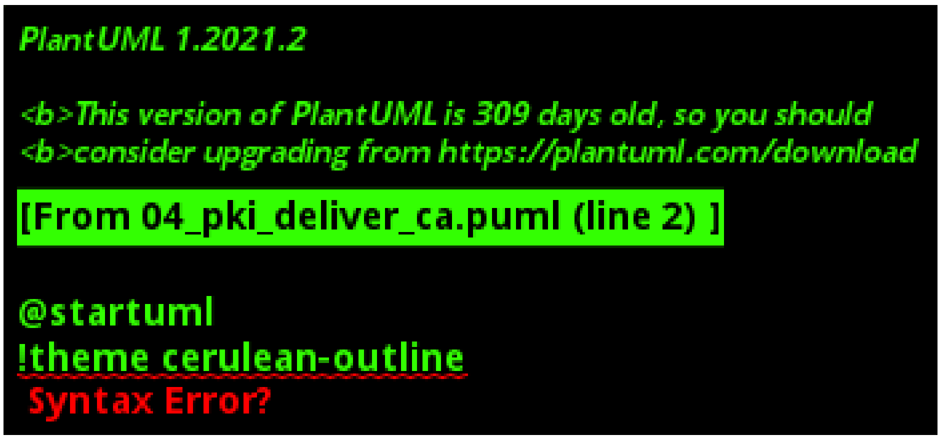
```
context.Header(
    "Content-Disposition",
    `attachment; filename="ca-cert.crt"`)
context.Data(
    http.StatusOK,
    "application/x-x509-ca-cert",
    certificates.GetCA())
```

The only specialty are 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”[28] encoded.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_pki_deliver_ca.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

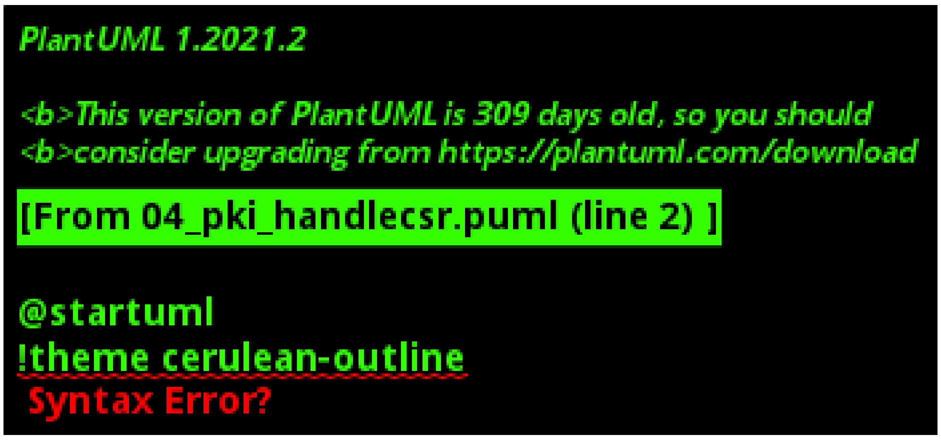
Figure 8: Deliver public certificate invocation

The process to deliver the CA is not very complex, as is shown in Figure 8. As soon as the startup process in Figure 7 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 6, 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.

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



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_pki_handlecsr.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 9: Invocation sequence to receive a signed certificate from the PKI.

The sequence in Figure 9 runs in the PKI. Since the certificate signing request is prepared with the private key of the translator (or the 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 exist. Since the current state of the system shall run within the same trust zone, this is not a big threat vector. However, for further implementations and iterations, a mechanism to "authenticate the authenticator" 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 measures 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[29].

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

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.

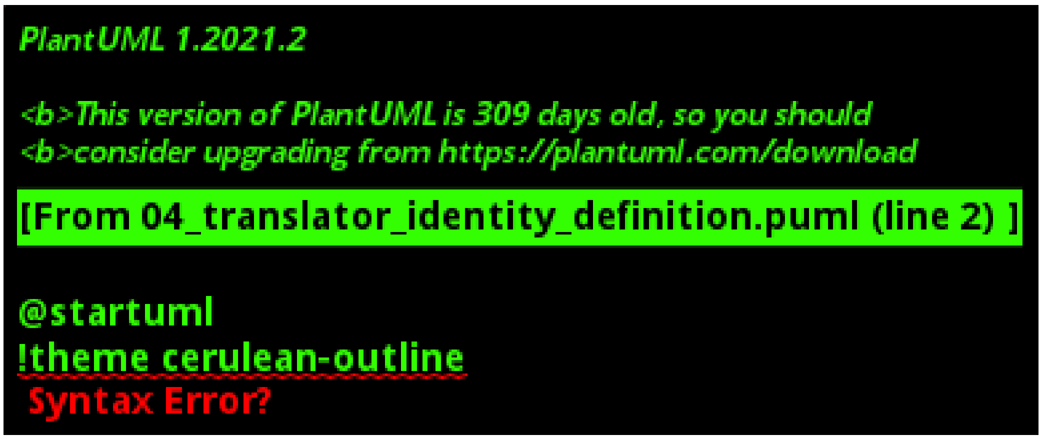
In either way, both measures require the usage of pre-shared or pre-known secrets. Additional options to mitigate this attack vector are not part of this project and shall be investigated in future work.

4.4 Provide a Translator Base

To enable developers to create translators easily, the GitHub organization "WirePact"[30] 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.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_translator_identity_definition.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

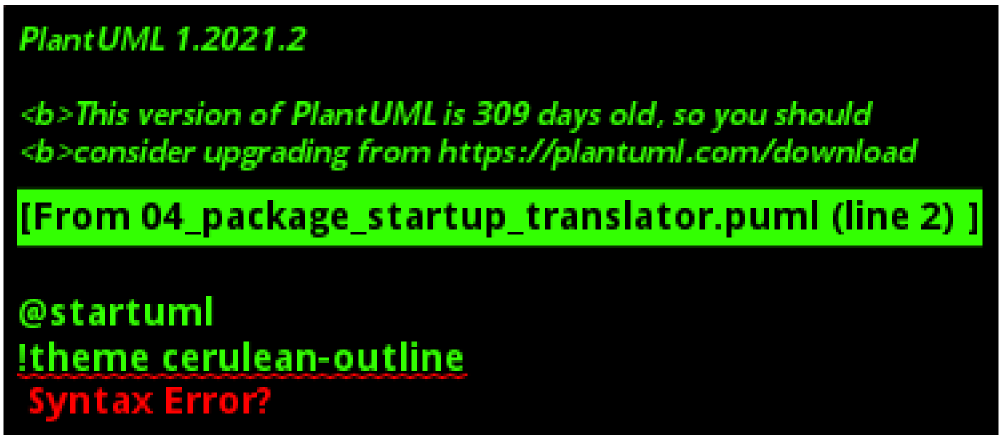
Figure 10: Definition of the Common Identity

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

As shown in Figure 10, 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 [12, 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.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_package_startup_translator.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 11: Startup Sequence of a Translator

Figure 11 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 privte 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
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.

```go
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 10), we encode it in the official specified JWT way and name it "sub" (i.e. "subject").

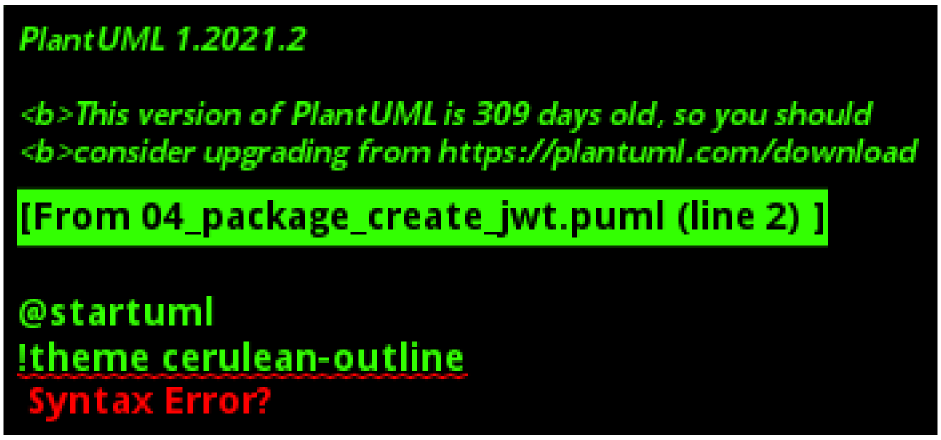
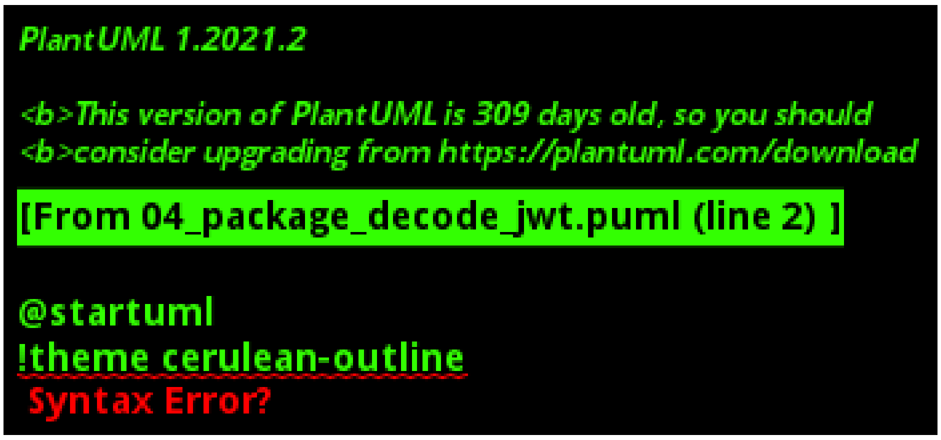

```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_package_create_jwt.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 12: Encode and Sign a JWT

The logic in Figure 12 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.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_package_decode_jwt.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 13: Decode JWT and Extract Subject

Figure 13 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 [13]. If a subject can be extracted, the developers code will be called and in the end, the request is forwarded if everything went fine.

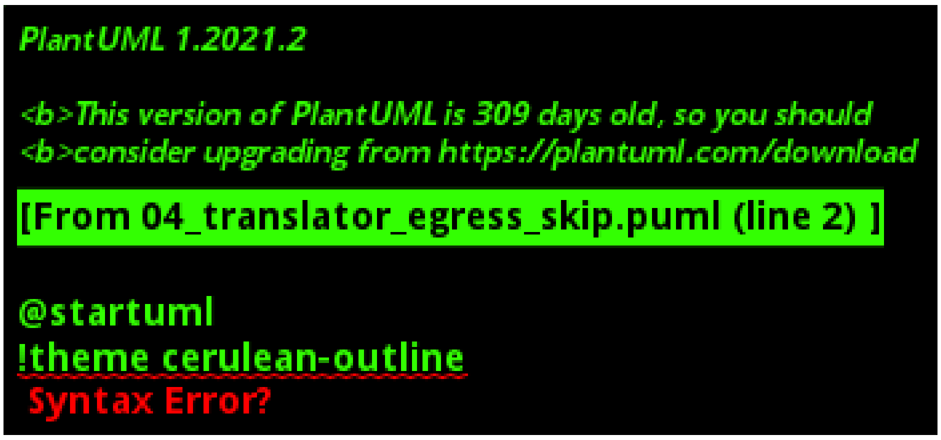
4.5 Implementing an HTTP Basic Translator with a Secure Common Identity

This section describes the definition and the usage of the secure common identity with the example of a translator. The translator uses HTTP Basic Auth (RFC7617 [6]) 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:

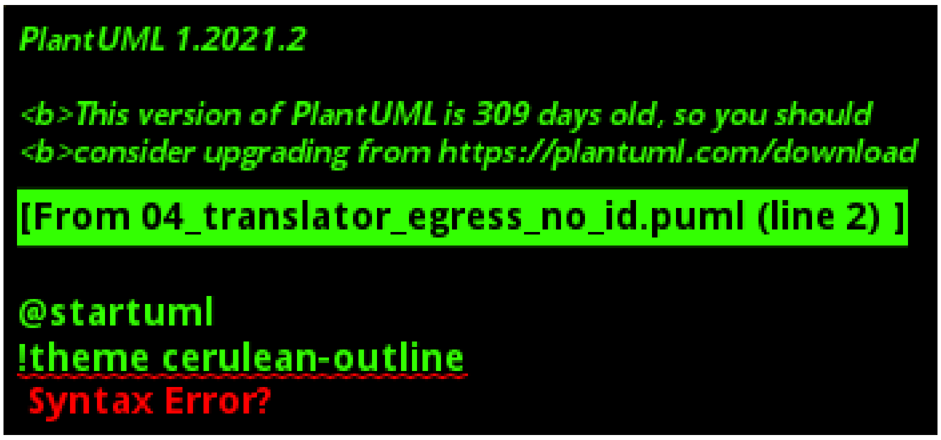
- Skip: Do not process any headers or elements in the request
- UserID empty: forbidden request
- Forbidden not empty: for some reason, the request is forbidden
- UserID not empty: the request is allowed



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_translator_egress_skip.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 14: Skipped/Ignored Egress Request

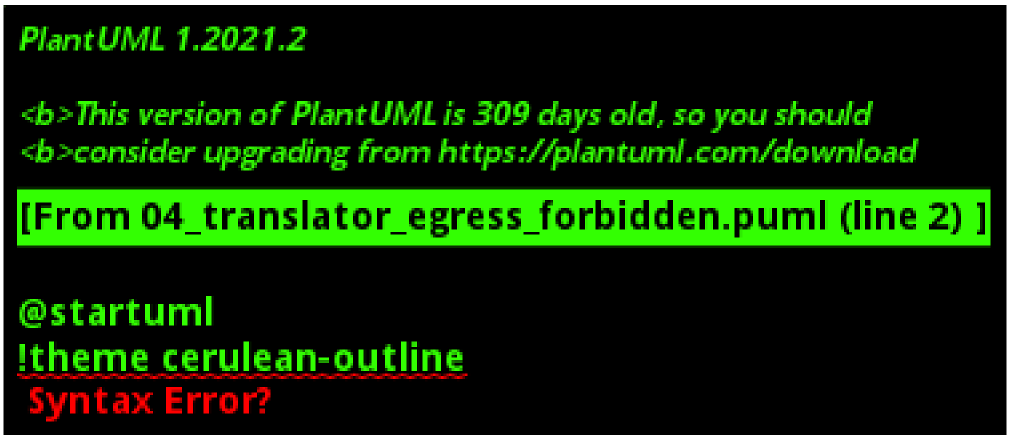
Figure 14 shows the sequence if the request is "skipped." In this case, skipped means that no headers are consumed nor added. The request is just passed to the destination without any interference. This happens if, in the case of Basic Auth, no `HTTP Authorize` header is added to the request or if another authentication scheme is used (OIDC for example).



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_translator_egress_no_id.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 15: Unauthorized Egress Request

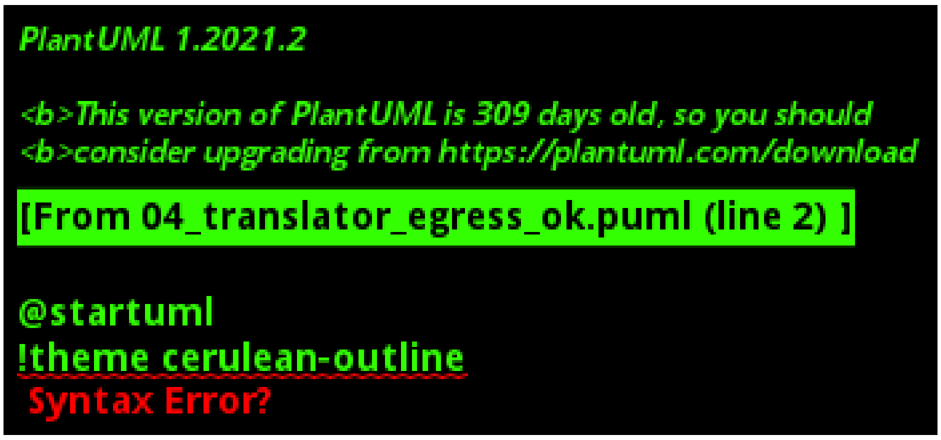
Figure 15 depicts the process when the request contains a correct HTTP header, but the provided username/password combination is not found in the "repository" of the translator. So, no common user ID can be found for the given username and therefore, the provided authentication information is not valid.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_translator_egress_forbidden.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 16: Forbidden Egress Request

In Figure 16, 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.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_translator_egress_ok.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

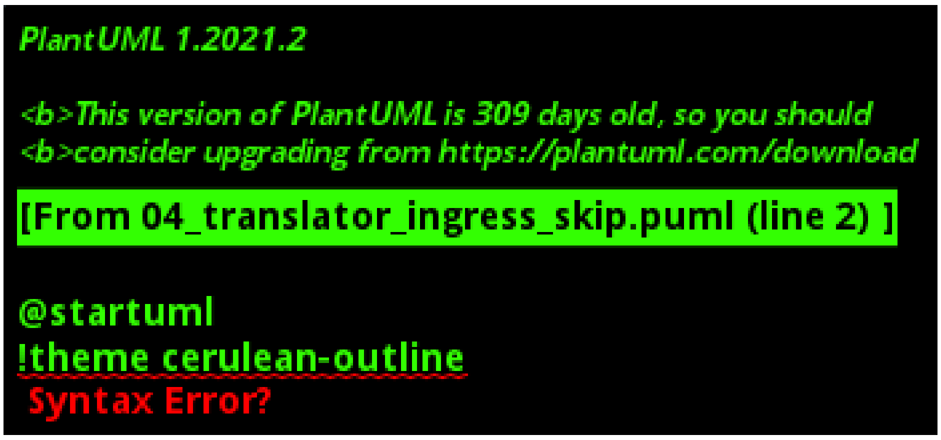
Figure 17: Processed Egress Request

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 17 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:

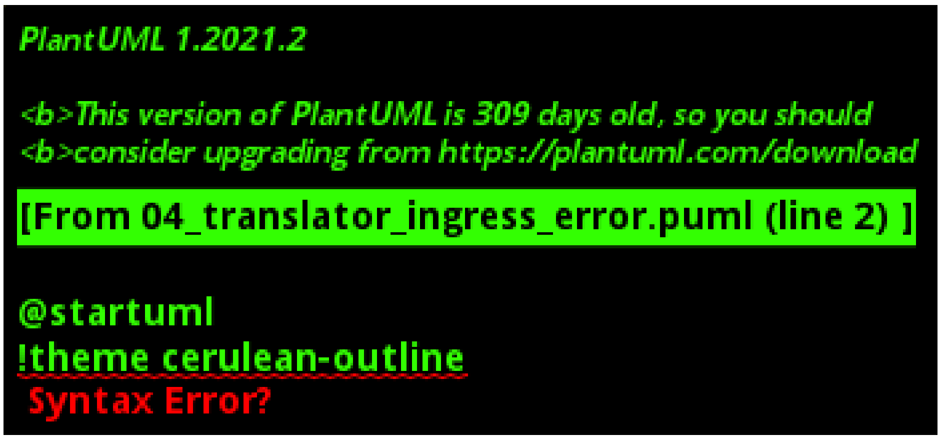
- Skip: if no information is given that relates to the authentication mesh.
- Error: if any error happens during validation.
- Forbidden: if the request is forbidden for any reason.
- OK: if the request is valid and the information about the user could be gathered.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_translator_ingress_skip.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 18: Skipped/Ingored Ingress Request

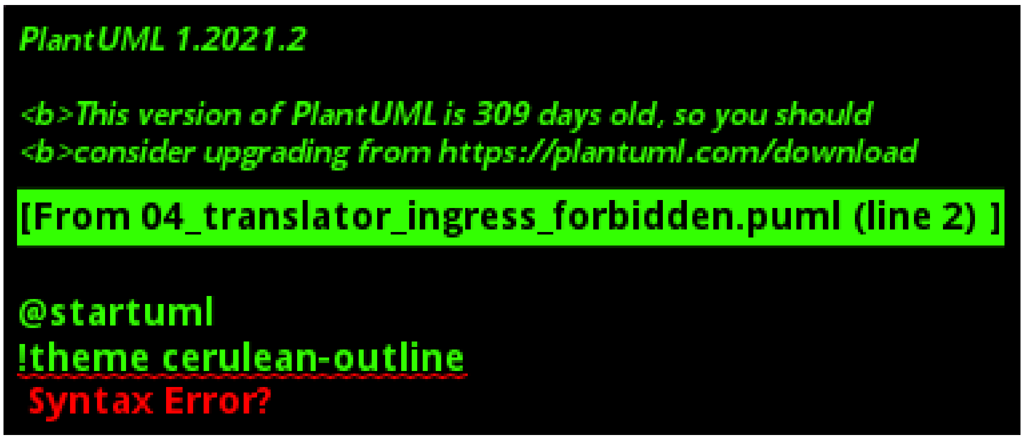
In Figure 18, the process for a neutral request is shown. The request contains no specific information that is relevant for the authentication mesh. Since the translator may not interfere with requests that are not “part of the mesh.” The request is skipped and the destination application may handle 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.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_translator_ingress_error.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 19: Errored Ingress Request

Error handling in the translator is bound to return forbidden responses. The translator should not throw any errors if possible. But if there are some errors, the translator returns a forbidden request to deny access to the destination as seen in Figure 19. If the translator would just skip the request, this could make the system vulnerable against error attacks.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_translator_ingress_forbidden.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 20: Forbidden Ingress Request

If the incoming request contains an `x-wirepact-identity` 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 20, the request is denied. No valid credentials mean that the translator cannot attach valid basic credentials for the target system.

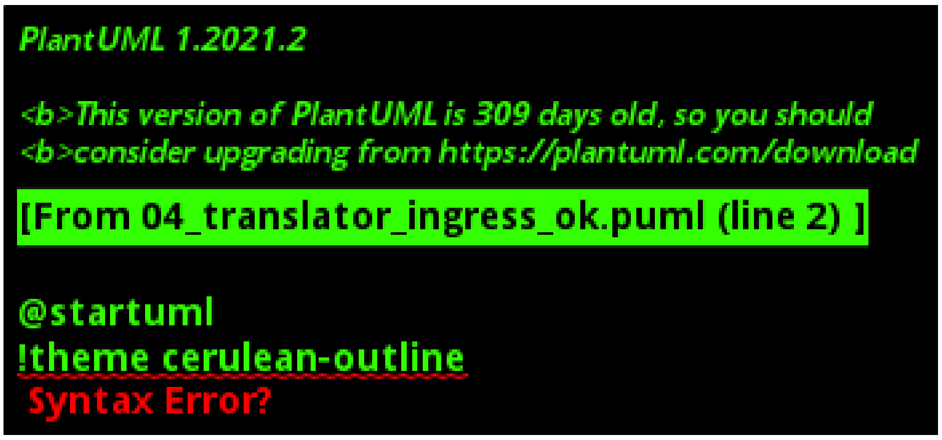

```
PlantUML 1.2021.2

<b>This version of PlantUML is 309 days old, so you should
<b>consider upgrading from https://plantuml.com/download

[From 04_translator_ingress_ok.puml (line 2) ]

@startuml
!theme cerulean-outline
 Syntax Error?
```

Figure 21: Successful Ingress Request

In contrast to the situations above, Figure 21 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.

4.6 Automate the Authentication Mesh

TODO: describe the operator

5 Conclusions and Outlook

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

[5] 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

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

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

[8] 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

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

[10] 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.

[11] 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.

[12] 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

[13] 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

Appendix A: Installation and Usage of the Mesh on Kubernetes

TODO: describe how the operator is installed in Kubernetes and how one can use the mesh