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 6, 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 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 parties 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 mentioned a "common language format" for the transport, but did not define nor implement it. 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 parts are still 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 and therefore 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.

TODO: sure? Additionally, it extends the concept of 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 custom logic with the help of a scripting language[3].

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[4] for the mesh. Furthermore, the evaluated pattern for the common language format is implemented in two different "translators" (HTTP Basic Auth and OpenID Connect). 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 thie 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]Such as Lua: https://www.lua.org/
[4]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 with Kubernetes. Kubernetes itself manages workloads and load balances them on several nodes (servers) while the patterns enable more complex applications and use-cases.

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.

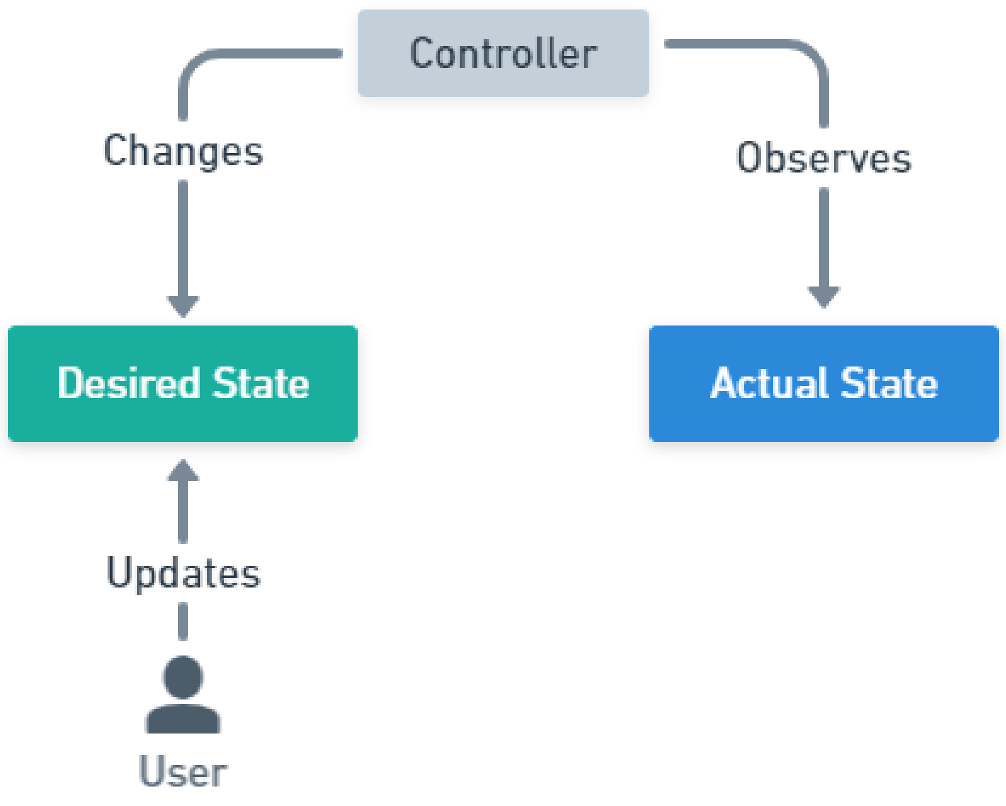


Figure 1: The Kubernetes Control Loop

Figure 1 shows the Kubernetes “control loop.” 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]. The custom extensions can be consumed by an Operator to manage complex applications. 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[5]: Manages instances of Prometheus (open-source monitoring and alerting software).
- Postgres Operator[6]: 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:
```

[5]https://github.com/prometheus-operator/prometheus-operator
[6]https://github.com/zalando/postgres-operator

```
    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`[7] 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[8].

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[9]: Go[10] Operator Framework
- KubeOps[11]: .NET Operator SDK
- Operator SDK[12]: SDK that supports Go, Ansible[13] or Helm[14]
- shell-operator[15]: 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.

2.2.3 A Sidecar, the Extension

Sidecars enhance a "Pod"[16] by injecting additional containers to the defined one [5].

[7] A form of deployment like `V1Deployment` but with certain stateful mechanics inside Kubernetes.
[8] Scraping: fetch the metrics from the target system and store them with time information
[9] https://book.kubebuilder.io/
[10] https://golang.org/
[11] https://buehler.github.io/dotnet-operator-sdk/
[12] https://operatorframework.io/
[13] https://www.ansible.com/
[14] https://helm.sh/
[15] https://github.com/flant/shell-operator
[16] The smallest possible workload unit in Kubernetes. A Pod contains of one or more containers that run a containerized application image.

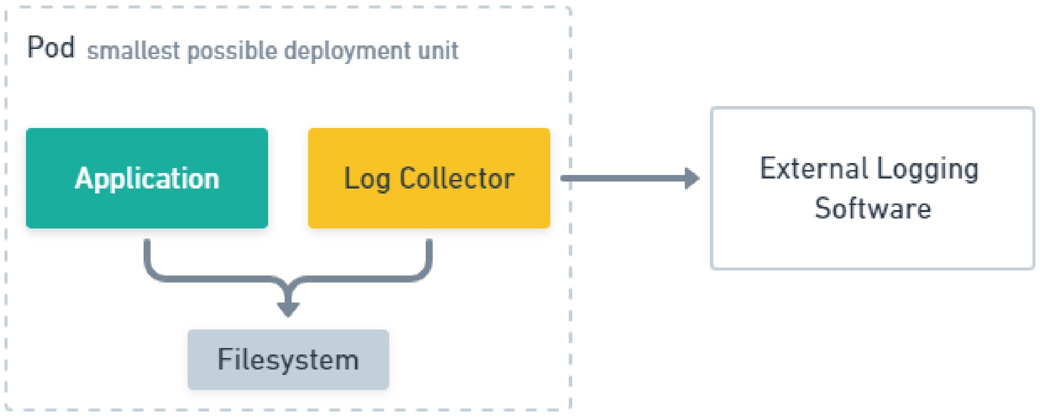

Pod smallest possible deployment unit
Application
Log Collector
Filesystem
External Logging Software

Figure 2: A Sidecar example

Figure 2 shows an example: A containerized application runs in its Docker[17] 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].

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` [6].

[17]https://www.docker.com/
[18]https://www.graylog.org/

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 [7].

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 [8]. 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 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.

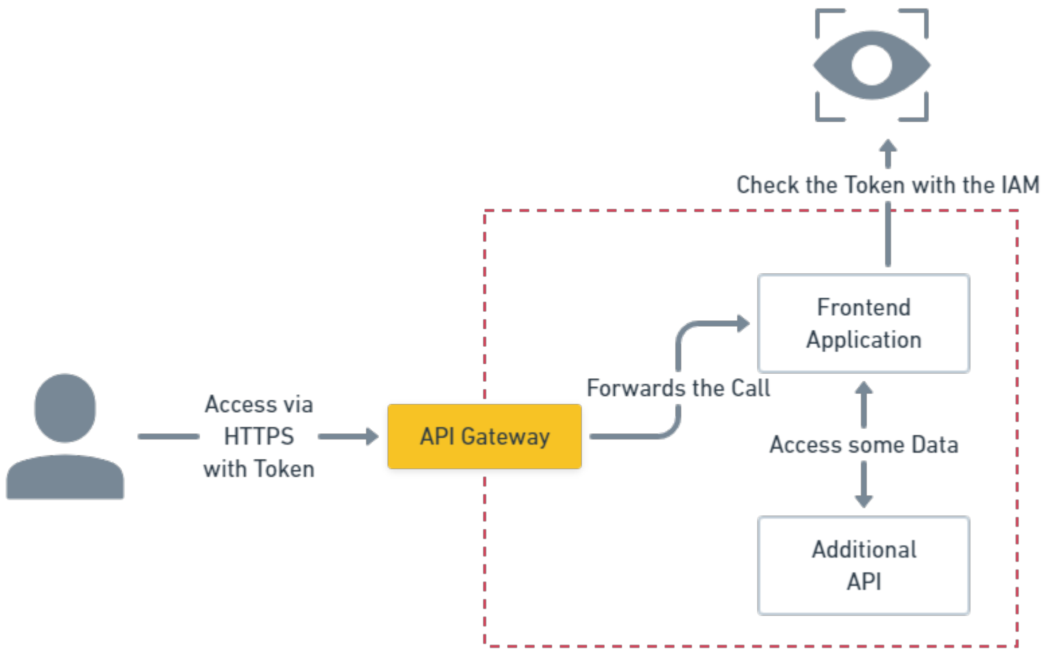


Figure 3: Example of a Trust Zone

As an example, we consider Figure 3. 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, 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 3 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[19] and applications to provide the safest possible implementations of such software.

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

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 yet. To enable the possibility of a production-grade software based on the concepts of the authentication mesh, this part must be specified.

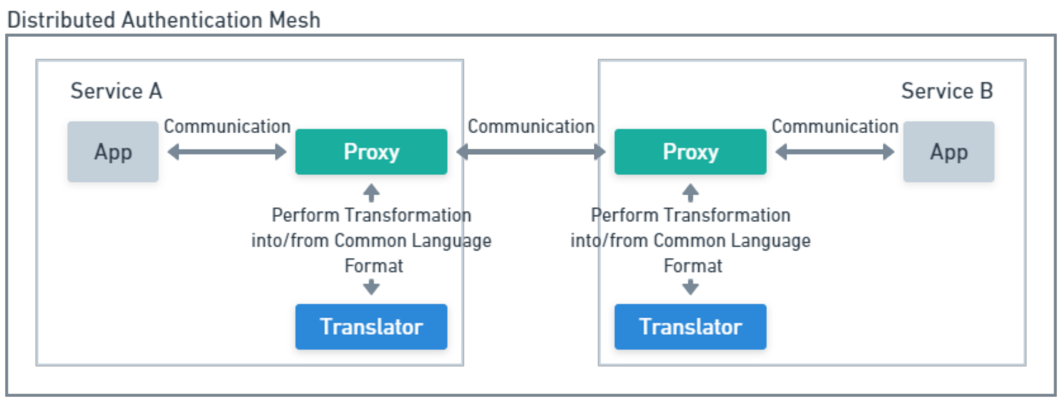

Distributed Authentication Mesh
Service A
App
Communication
Proxy
Perform Transformation into/from Common Language Format
Translator
Communication
Service B
Proxy
Communication
App
Perform Transformation into/from Common Language Format
Translator

Figure 4: General communication flow of two services in the distributed authentication mesh

Figure 4 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.

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[20], 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.

[20]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

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: TODO

- hashing/integrity check
- easy to use

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[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

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

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 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], seq. 4.2. 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"[24] 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[25] on the other hand, has a much better support for manipulating x509 certificates. Since the result of this project should

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

have a good developer experience, using x509 certificates is not be the best solution to solve the communication and integrity issue.

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 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 [RFC7515].

4.2.4 Using JWT in the Authentication Mesh

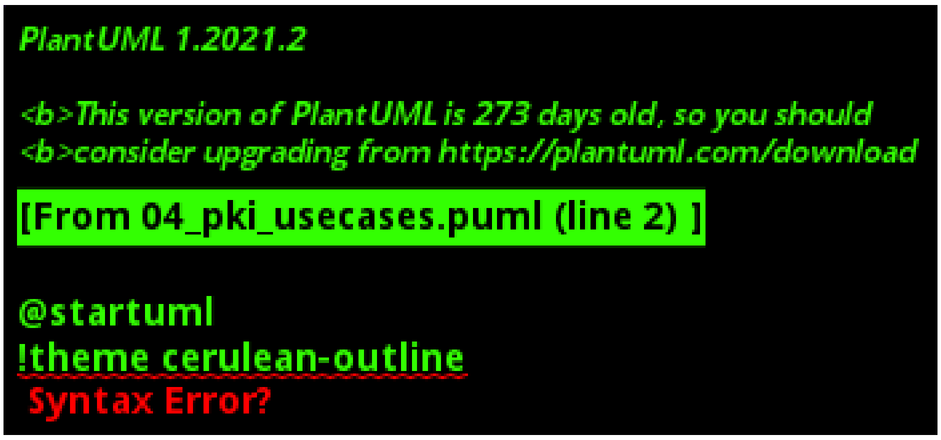
After considering the possible transport formats above, we can now analyze the pro and contra arguments. While **structured formats** like YAML and JSON are widely known and easily implemented, they do not offer a built-in mechanism to prevent data manipulation and integrity checking. There are standards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

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 5.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 273 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 5: Use Case Diagram for the PKI

Fetch CA Certificate. Any client 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`:

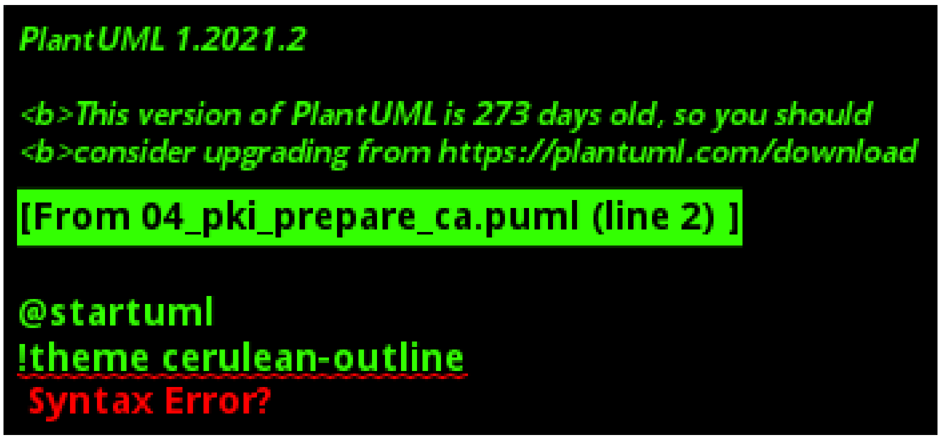
```
router := gin.Default()
```

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

```
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.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 273 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 6: 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 6 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 6 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.

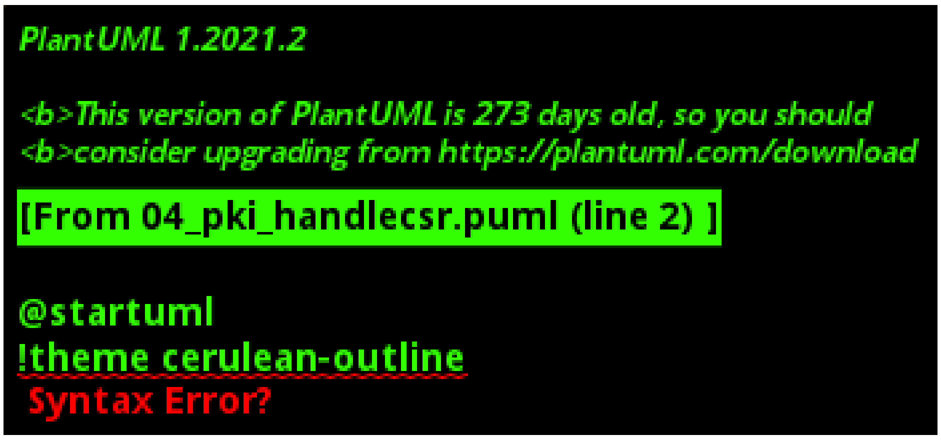
The `GetCA()` method itself just returns the public CA certificate:

```
func GetCA() []byte {
    return pem.EncodeToMemory(
        &pem.Block{Type: "CERTIFICATE", Bytes: ca.Raw})
}
```

The certificates are "PEM" encoded.

4.3.4 Process Certificate Signing Requests

To be able to sign CSRs, as stated in Figure 5, 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.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 273 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 7: Invocation sequence to receive a signed certificate from the PKI.

The sequence in Figure 7 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 are present. 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[27].

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.

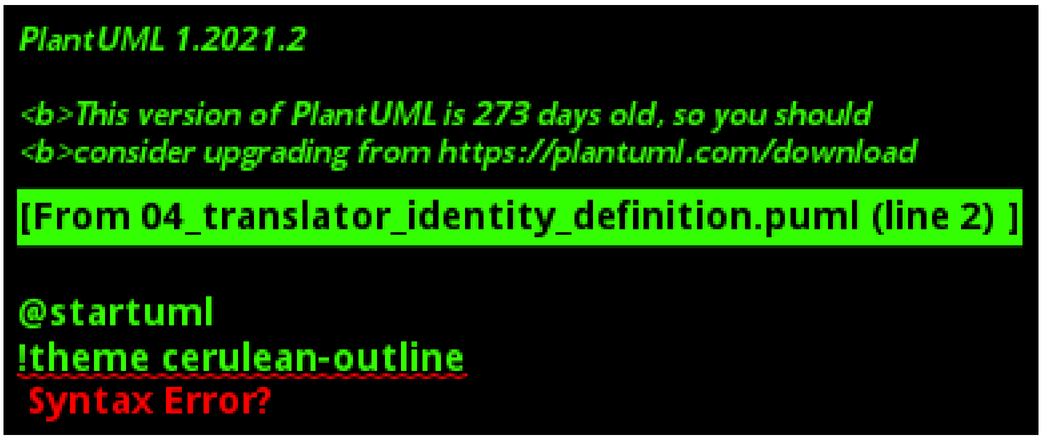
4.4 Implementing a Translator with a Secure Common Identity

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.

4.4.1 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.

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



```
PlantUML 1.2021.2

<b>This version of PlantUML is 273 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?
```

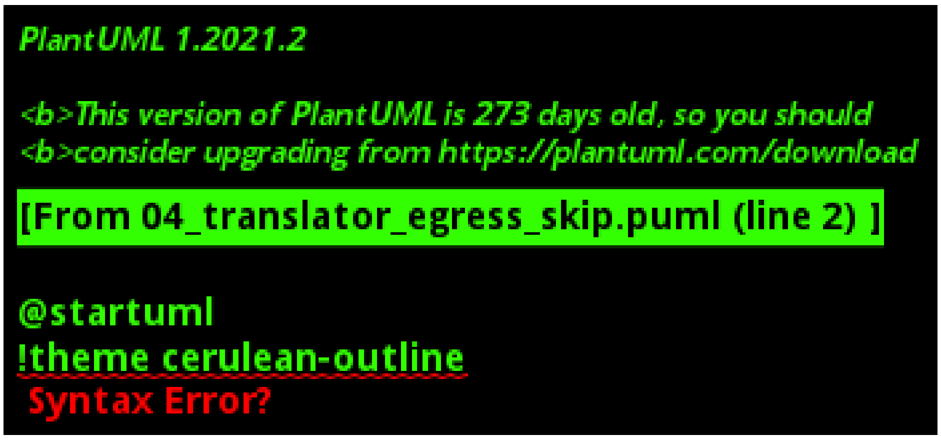
Figure 8: Definition of the Common Identity

As shown in Figure 8, 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].

4.4.2 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:

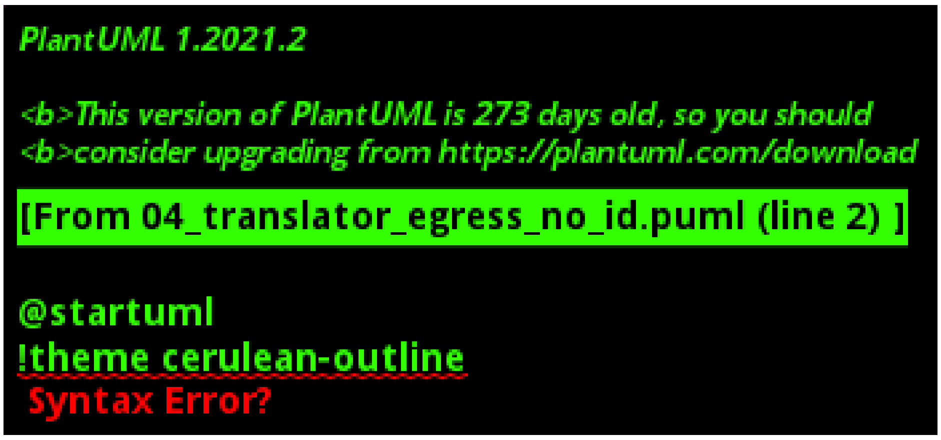
- Skip: Do not process any headers or elements in the request
- UserID empty: forbidden request
- Forbidden not empty: for some reason, the request is forbidden
- UserID not empty: the request is allowed



```
PlantUML 1.2021.2

<b>This version of PlantUML is 273 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 9: Skipped/Ignored Egress Request

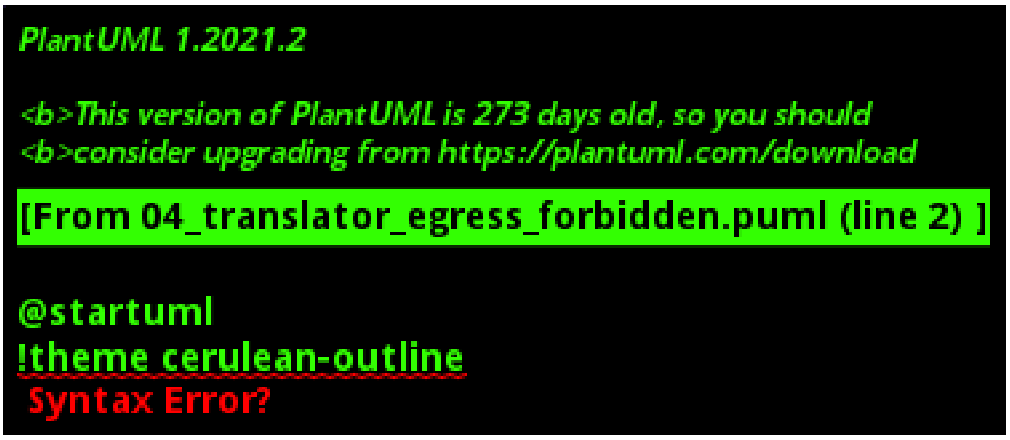
Figure 9 shows the sequence if the request is "skipped." In this case, skipped means that no headers are consumed nor added. The request is just passed to the destination without any interference. This happens if, in the case of Basic Auth, no `HTTP Authorize` header is added to the request or if another authentication scheme is used (OIDC for example).



```
PlantUML 1.2021.2

<b>This version of PlantUML is 273 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 10: Unauthorized Egress Request

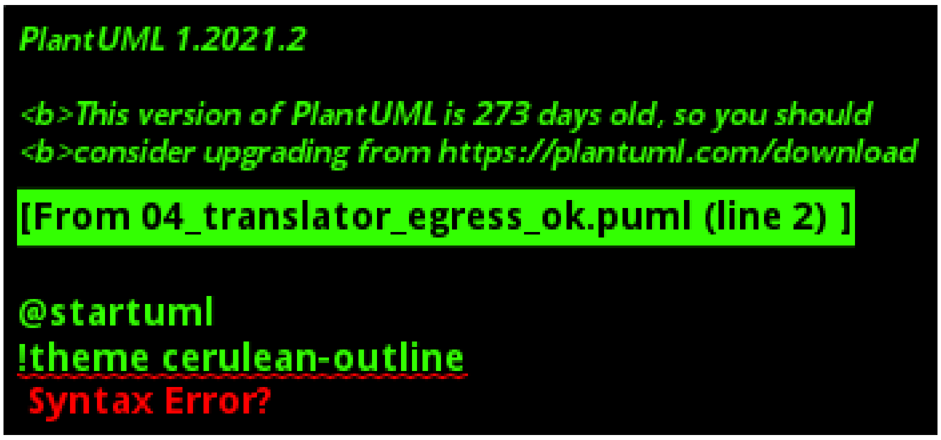
Figure 10 depicts the process when the request contains a correct HTTP header, but the provided username/password combination is not found in the "repository" of the translator. So, no common user ID can be found for the given username and therefore, the provided authentication information is not valid.



```
PlantUML 1.2021.2

<b>This version of PlantUML is 273 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 11: Forbidden Egress Request

In Figure 11, 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 273 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 12: 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 12 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.

4.4.3 Validate and Decode an Incoming Identity

4.5 Automate the Authentication Mesh

TODO: describe the operator

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] 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