Distributed Authentication Mesh*

Declarative Adhoc Conversion of Credentials

Christoph Bühler

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

*I would like to thank Dr. Olaf Zimmermann for reviewing and guiding this work. Furthermore, I thank Florian Forster, who acted as tester in the practice.

As more and more applications move to the cloud, security is a big concern. Applications shield themselves against attacks with various authentication mechanisms like OpenID Connect (OIDC). However, legacy appications that are not refactored nor updated tend to not support modern security standards. To enable applications to communicate with legacy (or third-party) software, it is often required to introduce code changes to the modern apps. To eliminate the leakage of credentials (like access tokens) and to reduce the risk of bugs, this project targets the dynamic conversion of a user identity. This identity authenticates the user over the wire instead of the effective credentials. This project gives the conceptional idea and the architecture, as well as a platform specific example of such a solution. Furthermore, a proof of concept answers relevant questions for the realization of such a framework. The evaluation then validates the architecture against the goals and non-goals and checks the architecture. In the end, the conclusion provides information about the project, possible further work, and the goals of the next project-phase.

Contents

List of Tables

List of Figures

1 Introduction

Modern cloud environments solve many problems like the discovery of services and data transfer or communication between services in general. As the development of cloud-native applications (CNA) evolves, older applications move to the cloud as well.

However, a specific problem is not yet solved: "dynamic" trusted communication between services. For example, a service that is capable of handling OpenID Connect (OIDC) credentials wants to communicate with a service that only knows Basic Authentication. The source service must implement some conversion mechanism or know static credentials to communicate with the basic auth service. In general, this introduces changes to the software. In small applications which consist of one or two services, implementing this conversion may be a feasible option. However, if we look at an application that spans over a big landscape and a multitude of services, implementing every possible authentication mechanism and the according to conversions will be error-prone work and does not scale well[1]. In practice, we encountered the given scenario at various points in time when older applications were migrated into a cloud environment and newer applications were built around it. In almost all cases, the modern software was changed to communicate with the legacy systems, not the other way around.

The goal of the project "Distributed Authentication Mesh" is to provide a solution for this exact problem. By introducing multiple elements, like a translator in conjunction with a proxy that is capable of modifying HTTP headers in-flight, the described problem can be solved. The proposed concept uses a common domain language to transfer the authenticated identity of a user between services of a microservice application. The proxy then intercepts the requests and instructs the translator to transform the common language into the valid authentication format of the destination.

For further reading, basic knowledge about "Docker" and microservices is required. Furthermore, the implementation of the proof of concept (POC) is based on "Kubernetes" to display the concepts of the solution in a practical manner. In terms of authentication and authorization, the POC uses OpenID Connect and Basic Authentication, which are both described in later sections.

The remainder of the report describes used technologies, terminology, and concepts. Furthermore, the state of the art gives an overview of the current situation and the present solutions in practice. With the description of the distributed authentication mesh, the report shows the proposed solution and the architecture. Then, some elements

[1] According to the matrix problem: X services $* Y$ authentication methods

of the solution are tested with an implementation of a POC on the Kubernetes platform. After the description of the solution, the evaluation checks if the goals and non-goals of the solution are valid. The conclusion then gives an overview of the work and a summary of the project.

2 Definitions and Clarification of the Scope

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

2.1 Scope of the Project

This project addresses the specific problem of declarative conversion of user credentials, for example an access token, to ensure authorized communication between services. When multiple services with different authentication mechanisms communicate with each other, the services need to translate the credentials and send them to their counterpart. The goal of this project is to prevent user credentials from being transmitted to other services.

To solve this problem, an automation component enhances services that are part of the application with additional functionality. A proxy in front of the service captures in-, and outgoing traffic to modify the `Authorization` HTTP header. Additionally, a translator transforms the original authentication data into a form of identity and encodes it with a common language format. The receiving service can validate this encoded identity and transforms the identity into valid user credentials again. This automatical transformation of credentials (e.g. from OIDC to Basic Auth) replaces manual work which may introduce code changes to either service. The deliverables of this and further projects may aid applications or APIs to communicate with each other despite different authentication mechanisms.

The solution may be runnable on various platforms but to provide a practical demo application, the proof of concept (POC) runs on Kubernetes. Kubernetes[1] is an orchestration platform that works with containerized applications. The POC resides in an initial version in an open-source GitHub repository. The POC demonstrates that it is possible to instruct an Envoy[2] proxy to communicate with an injected service to modify authentication credentials in-flight. To separate the proposed solution from complexer concepts like a service mesh, the POC is able to run without a service mesh on a Kubernetes cluster and uses the built in service discovery of Kubernetes to communicate.

[1] https://kubernetes.io/
[2] https://www.envoyproxy.io/

2.2 Kubernetes as an Orchestration Engine

This section provides a general overview of Kubernetes. Kubernetes is a prominent orchestration engine that manages workloads on worker-nodes.

2.2.1 Introduction

Kubernetes is an open-source platform that manages containerized workloads and applications. Workloads may be accessed via "Services" that use a DNS naming system. Kubernetes uses declarative definitions to compare the actual state of the system with the expected state (CNCF 2021).

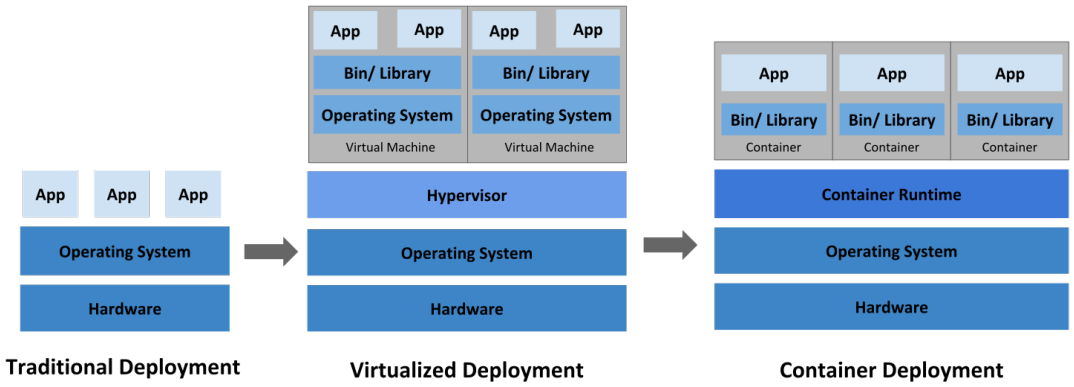

Figure 2.1: Container and Deployment Evolution. Description of the evolution of deployments as found on the documentation website of Kubernetes (CNCF 2021). This image is licensed under the CC BY 4.0 license (Creative Commons 2021).

According to Kubernetes, the way of deploying applications has evolved. As shown in Figure 2.1, the "Traditional Era" was the time when applications were deployed via FTP access and started manually (e.g. on an Apache webserver). Then the revolution to virtual machines came and technologies that could virtualize a whole operating system, such as VMWare, were born. The latest stage, "Container Era," defines a new way deploying workloads by virtualizing processes instead of operating systems and therefore better use the given resources (CNCF 2021).

Kubernetes is a major player among others like "OpenShift" or "Cloud Foundry" in "Container Deployment" as seen in Figure 2.1 and supports teams with the following features according to the documentation (CNCF 2021):

- **Service discovery and load balancing**: Use DNS names or IP addresses to route traffic to a container and if the traffic is high and multiple instances are available, Kubernetes does load balance the traffic

- **Storage orchestration**: Automatically provide storage in the form of mountable volumes
- **Automated rollouts and rollbacks**: When a new desired state is provided Kubernetes tries to achieve the state at a controlled rate and has the possibility of performing rollbacks
- **Automatic bin packing**: Kubernetes only needs to know how much CPU and RAM a workload needs and then takes care of placing the workload on a fitting node in the cluster
- **Self-healing**: If workloads are failing, Kubernetes tries to restart the applications and even kills services that do not respond to the configured health checks
- **Secret and configuration management**: Kubernetes has a store for sensitive data as well as configurational data that may change the behavior of a workload

The list of features is not complete. There are many concepts in Kubernetes that help to build complex deployment scenarios and enable teams to ship their applications in an agile manner.

Kubernetes works with containerized applications. In contrast to "plain" Docker, it orchestrates the applications and is responsible to achieve the desired state depicted in the application manifest files. Examples of such deployments and other Kubernetes objects are available online in the documentation (CNCF 2021)[3].

2.2.2 Terminology

In Table 2.1, we state the most common Kubernetes terminology. The table provides a list of terms that is used to explain concepts like the operator pattern in Section 2.3.

Table 2.1: Common Kubernetes Terminology

Term	Description
Docker	Container runtime. Enables developers to create images of applications. Those images are then run in an isolated environment. Docker images are often used in Kubernetes to define the application that Kubernetes should run.
Kustomize	"Kustomize" is a special templating CLI to declaratively bundle Kubernetes manifests. It consists of a `kustomization.yaml` and various referenced manifest `yaml` files. It is declarative and does not allow dynamic structures. It helps administrators to template applications for Kubernetes.
Container	Smallest possible unit in a deployment. Contains the definition of the workload. A container consists of a container image, arguments, volumes and other specific information to carry out a task.

[3]https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment

Term	Description
Pod	Composed of multiple containers. Is ran by Kubernetes as an instance of a deployment. Pods may be scaled according to definitions or "pod scalers." Highly coupled tasks are deployed together in a pod (i.e. multiple coupled containers in a pod).
Deployment Daemonset Statefulset	A deployment is a managed instance of a pod. Daemonsets and Statefulsets are variants of deployments. Kubernetes will run the described pod with the desired replica count on the best possible worker node. Deployments may be scaled with auto-scaling mechanisms.
Service	A service enables communication with one or multiple pods. The service contains a selector that points to a certain number of pods and then ensures that the pods are accessible via a DNS name. The name is typically a combination of the service name and the namespace (e.g. `my-service.namespace`).
Ingress	Incoming communication and data flow into a component. Furthermore an "Ingress" is a Kubernetes object that defines incoming communication and configures an API gateway to route traffic to specific services.
Egress	Outgoing communication. Egress means communication from a component to another (when the component is the source).
Resource	A resource is something that can be managed by Kubernetes. It defines an API endpoint on the master node and allows Kubernetes to store a collection of such API objects. Examples are: `Deployment`, `Service` and `Pod`, to name a few of the built-in resources.
CRD	A Custom Resource Definition (CRD) enables developers to extend the default Kubernetes API. With a CRD, it is possible to create own resources which create an API endpoint on the Kubernetes API. An example of such a CRD is the `Mapping` resource of Ambassador[4].
Operator	An operator is a software that manages Kubernetes resources and their lifecycle. Operators may use CRDs to define custom objects on which they react when some event (`Added`, `Modified` or `Deleted`) triggers on a resource. For a more in-depth description, see Section 2.3.
Watcher	A watcher is a constant connection from a client to the Kubernetes API. The watcher defines some search and filter parameters and receives events for found resources.
Validator	A validator is a service that may reject the creation, modification or deletion of resources.

Term	Description
Mutator	Mutators are called before Kubernetes validates and stores a resource. Mutators may return JSON patches **RFC6902** (Bryan and Nottingham 2013) to instruct Kubernetes to modify a resource prior to validating and storing them.

TODO: UML of kubernetes parts

2.3 The Operator Pattern

An operator in Kubernetes is an extension to the Kubernetes API itself. A custom operator typically manages the whole lifecycle of an application it manages (Dobies and Wood 2020). Such a custom operator can further be used to reconcile normal Kubernetes resources or any combination thereof.

Some examples of application operators are:

- Prometheus Operator[5]: Manages instances of Prometheus in a cluster
- Postgres Operator[6]: Manages PostgreSQL clusters inside Kubernetes, with the support of multiple instance database clusters

There exists a broad list of operators, which can be (partially) viewed on https://operatorhub.io.

In Figure 2.2, we depict the general workflow of an event that is managed by an operator. When an operator is installed and runs on a Kubernetes cluster, it registers "Resource Watchers" with the API and receives notifications if the master node modifies resources. The overviewed events are "Added," "Modified" and "Deleted." There are two additional events that may be returned by the API ("Error" and "Bookmark"), but they are typically not needed for reconciliation.

When the user interacts with the Kubernetes API (e.g. via the `kubectl` executable) and creates a new instance of a resource, the API will first call any "Mutator" in a serial manner. After the mutators, the API will call any "Validators" in parallel and if no validator objects against the creation, the API will then store the resource and tries to apply the transition for the new desired state. Now, the operator receives a notification about the watched resource and may interact with the event. Such an action may include to update resources, create more resources or even delete other instances.

[4]https://www.getambassador.io/
[5]https://github.com/prometheus-operator/prometheus-operator
[6]https://github.com/zalando/postgres-operator

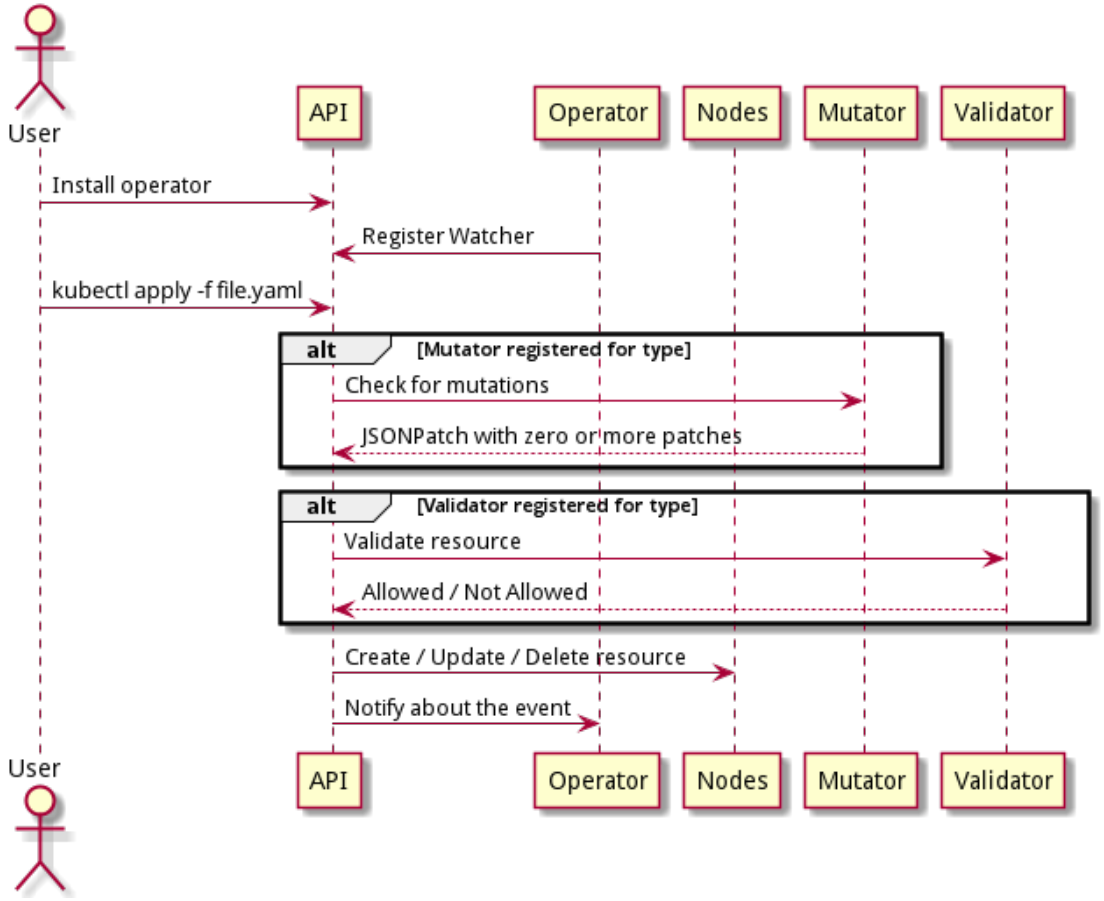

Figure 2.2: Kubernetes Operator Workflow

2.4 The Sidecar Pattern

According to Burns and Oppenheimer (2016), the sidecar pattern is the most common pattern for multi-container deployments (Burns and Oppenheimer 2016, sec. 4.1). Sidecars are containers that enhance the functionality of the main container in a pod. An example for such a sidecar is a log collector, that collects log files written to the file system and forwards them towards some log processing software (Burns and Oppenheimer 2016, sec. 4.1). Another example is the Google CloudSQL Proxyˆ[https://github.com/GoogleCloudPlatform/cloudsql-proxy, which provides access to a CloudSQL instance from a pod without routing the whole traffic through Kubernetes services.

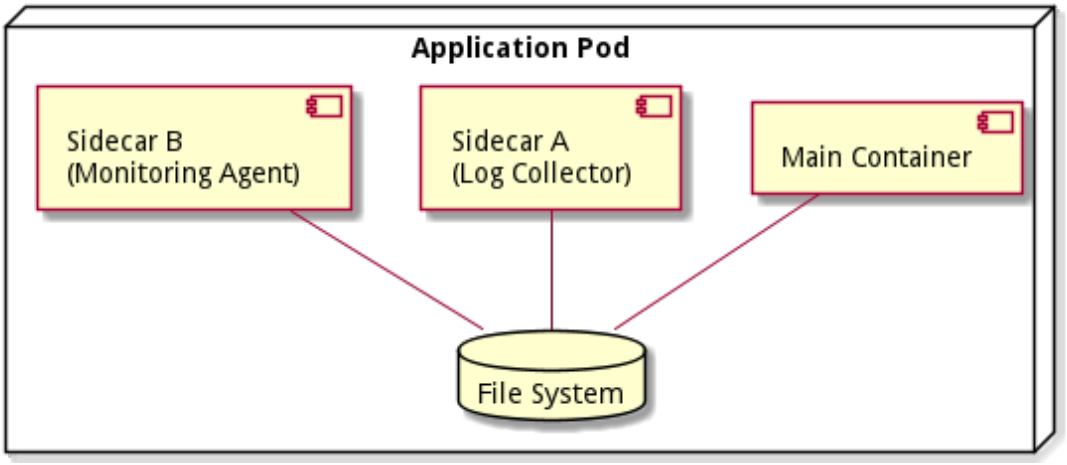

Figure 2.3: Sidecar container extending a main container in a pod. As example, this could be a log collector (Burns and Oppenheimer 2016, fig. 1).

The example shown in Figure 2.3 is extensible. Such sidecars may be injected by a mutator or an operator to extend functionality.

Common use cases for sidecars include controlling the data flow in a cluster in service mesh[7], providing access to secure locations[8] or performing additional tasks such as collecting logs of an application. Since sidecars are tightly coupled to the original application, they scale with the pod. It is not possible to scale a sidecar without scaling the pod - and therefore the application - itself.

2.5 Controlling the Data with a Service Mesh

A “Service Mesh” is a dedicated infrastructure layer that handles intercommunication between services. It is responsible for the delivery of requests in a modern cloud application

[7] As done by istio (https://istio.io/latest/docs/reference/config/networking/sidecar/)
[8] Like the Google CloudSQL Proxy

(Li et al. 2019, sec. 2). An example from the practice is "Istio"[9]. When using Istio, the applications do not need to know if there is a service mesh installed or not. Istio will inject a sidecar (see Section 2.4) into the deployments to handle the communication between services.

The service mesh provides a set of features (Li et al. 2019, sec. 2):

- **Service discovery**: The mechanism to locate and communicate with a workload / service. In a cloud environment, the location of services will likely change, thus the service mesh provides a way to access the services in the cloud.
- **Load balancing**: As an addition to the service discovery, the mesh provides load balancing mechanisms as is done by Kubernetes itself.
- **Fault tolerance**: The router in a service mesh is responsible to route traffic to healthy services. If a service is unavailable or even reports a crash, traffic should not be routed to this instance.
- **Traffic monitoring**: In contrast to the default Kubernetes possibilities, with a service mesh, the traffic from and to various services can be monitored in detail. This offers the opportunity to derive reports per target, success rates and other metrics.
- **Circuit breaking**: The ability to cut off an overloaded service and back off the remaining requests instead of totally failing the service under stress. A circuit breaker pattern measures the failure rate of a service and applies states to the service: "Closed" - requests are passed to the service, "Open" - requests are not passed to this instance, "Half-Open" - only a limited number is passed (Montesi and Weber 2016, sec. 2).
- **Authentication and access control**: Through the control plane, a service mesh may define the rules of communication. It defines which services can communicate with one another.

As observed in the list above, many of the features of a service mesh are already provided by Kubernetes. Service discovery, load balancing, fault tolerance and - though limited - traffic monitoring is already possible with Kubernetes. Introducing a service mesh into a cluster enables administrators to build more complex scenarios and deployments.

2.6 Authentication, Authorization, and Security

This section provides an introduction in the used authentication mechanisms. The proposed solution is capable of handling more than the described schemes but for the implementation of the POC, Basic Authentication and OIDC were used.

[9] https://istio.io/

2.6.1 Basic Authentication (RFC7617)

The `Basic` authentication is a trivial authentication scheme (i.e. a way to prove the identity of an entity) that accepts a username and a password encoded in Base64. To transmit the credentials, a construct with the schematics of `<username>:<password>` is created and inserted into the HTTP request as the `Authorization` header with the prefix `Basic` (Reschke 2015, sec. 2). An example with the username `ChristophBuehler` and password `SuperSecure` would result in the following header:

```
Authorization: Basic Q2hyaXN0b3BoQnVlaGxlcjpTdXBlclNlY3VyZQ==
```

2.6.2 OpenID Connect (OIDC)

OpenID Connect is not defined in a RFC, the specification is provided by the OpenID Foundation (OIDF). OIDC extends OAuth, which is defined by **RFC6749**.

OpenID Connect is an authentication scheme, that extends the `OAuth 2.0` framework. OAuth itself is an authorization framework, that enables applications to gain access to a resource (API or other) (Hardt and others 2012, abstract). OAuth 2.0 only deals with authorization and grants access to data and features on a specific application. OAuth by itself does not define *how* the credentials are transmitted and exchanged (Hardt and others 2012). OIDC adds additional logic to OAuth 2.0 that defines *how* these credentials must be exchanged. Thus, OIDC enables login and profile capabilities in any application that uses OIDC (Sakimura et al. 2014).

When a user wants to authenticate himself with OIDC, one of the possible "flows" is the "Authorization Code Flow." Other possible flows are the "Implicit Flow" and the "Hybrid Flow" (Sakimura et al. 2014, sec. 3.1, sec. 3.2, sec. 3.3). Figure 2.4 depicts the "Authorization Code Flow." A user that wants to access a certain resource (e.g. an API) on a relying party (i.e. something that relies on the information about the user) and is not authenticated and authorized, the relying party forwards the user to the identity provider (IdP). The user provides his credentials to the IdP and is returned to the relying party with an authorization code. The relying party can then exchange the authorization code to valid tokens on the token endpoint of the IdP. Typically, `access_token` and `id_token` are provided. While the `id_token` must be a JSON Web Token (JWT), the `access_token` can be in any format (Sakimura et al. 2014, sec. 2, sec. 3.3.3.8).

An example of an `id_token` in JWT format may be:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiw
```

The stated JWT token contains:

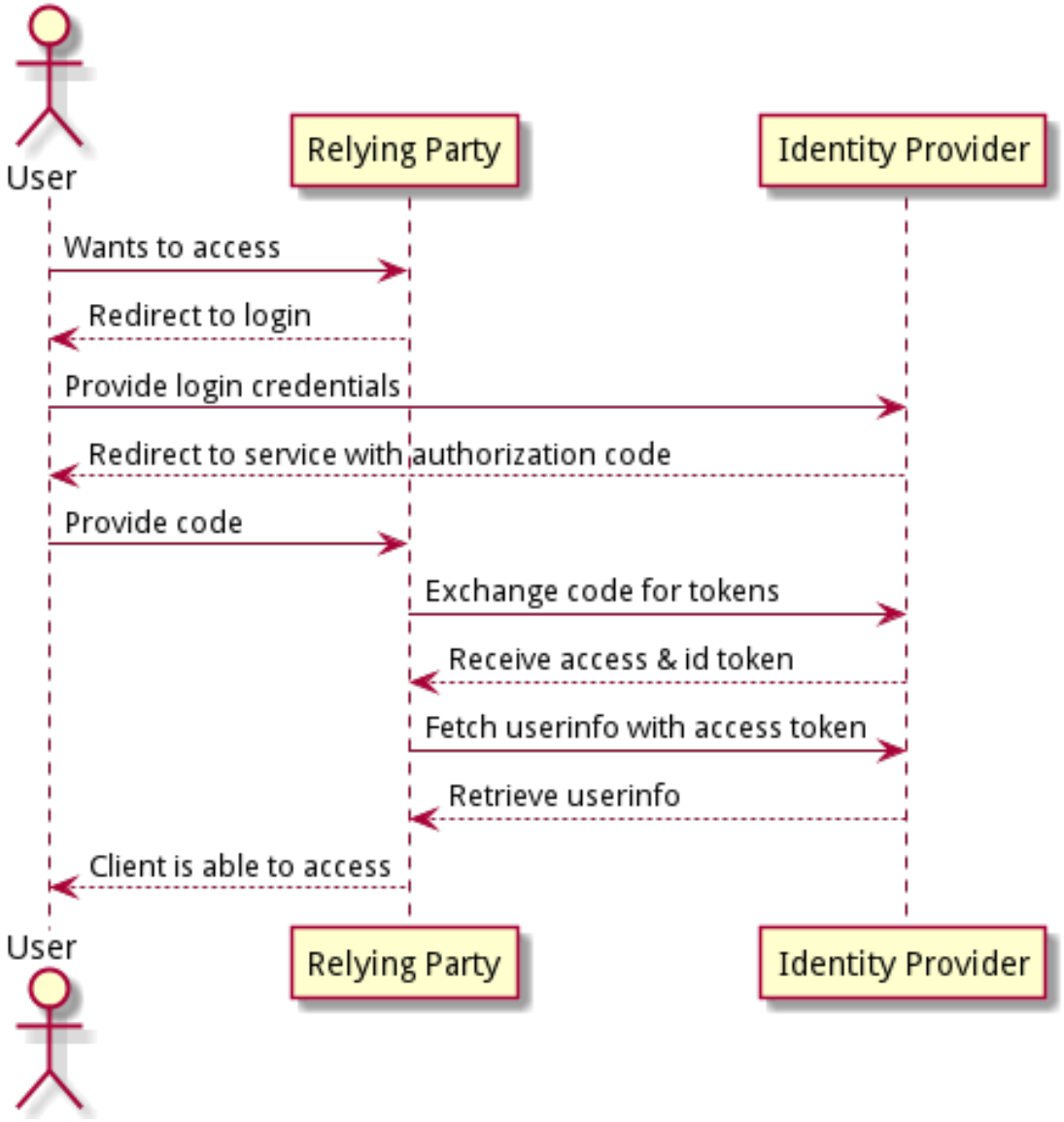

Figure 2.4: OIDC code authorization flow (Sakimura et al. 2014). Only contains the credential flow, without the explicit OAuth part. OAuth handles the authorization whereas OIDC handles the authentication.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

2.6.3 Zero Trust Environment

"Zero Trust" is a security model with its focus on protecting data and user credentials. The basic idea of zero trust is to assume that an attacker is always present. It does not matter if the application resides within an enterprise network, zero trust assumes that enterprise networks are no more trustworthy than any other public network. As a consequence of zero trust, applications are not implicitly trusted. Therefore, user credentials must be presented and validated for each access to a resource (Rose et al. 2019). Zero trust can be summarized with: "Never trust, always verify."

3 State of the Art and the Practice

This section gives an overview of the current state of the art and the practice. Furthermore, it states the deficiencies that this project tries to solve.

3.1 Accessing Legacy Systems from Modern Systems

In cloud environments, a solved problem is the transmission of arbitrary data from one endpoint to another. Modern programming languages (like .NET, Python and Node.js) provide ways to handle communication with other endpoints and APIs. To transmit data between services in a cloud environment, an application can use the HTTP protocol or gRPC[1] to encode the requests and responses in a common format. In the case of a service mesh, a sidecar is injected into the pod that contains a proxy to handle data transmission between the services.

In terms of authentication and authorization, there is a variety of schemes that enable an application to authenticate and authorize its users. OpenID Connect (OIDC) (see Section 2.6.2) is a modern authentication scheme, that extends the OAuth 2.0 framework, which in turn handles authorization (Sakimura et al. 2014). OAuth only defines how to grant access to specific resources (like APIs) but not how they are exchanged. OIDC fills that space by introducing authentication flows (e.g. "Authorization Code Flow" in Figure 2.4). OAuth in combination with OIDC provides a modern and secure way of authentication and authorizing users against an API.

Modern software architectures that are specifically designed for the cloud are called "Cloud Native Applications" (CNA). Kratzke and Peinl (2016) define a CNA as:

> "A cloud-native application is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform."

However, with CNAs and the general movement to cloud environments and digitalization, not all applications get that chance to adjust. For various reasons like budget, time or technical risks and skill availability, legacy applications and monoliths are not always

[1] https://grpc.io/

refactored or re-written before they are deployed into a cloud environment. If the legacy applications (for example an old ERP system) are mixed with modern systems, then the need of "translation" arises. Assuming that the modern part is a secure application that uses OIDC to authenticate its users and the application needs to fetch data from a legacy system. The legacy application does not understand OIDC, thus either the modern or the legacy application must receive code changes (i.e. enable the application to convert the user credentials to the scheme of the target service) to enable communication between the services. Following the previous assumption, the code changes will likely be introduced into the modern application, since it is presumably better maintainable and deployable than the legacy app. Hence, the modern application receives changes that may introduce new bugs or vulnerabilities. If new code is introduced into an application, "normal" software bugs may be created and external dependencies (such as libraries for authentication and authorziation) may import vulnerabilities caused by bugs or by deviation from the standards.

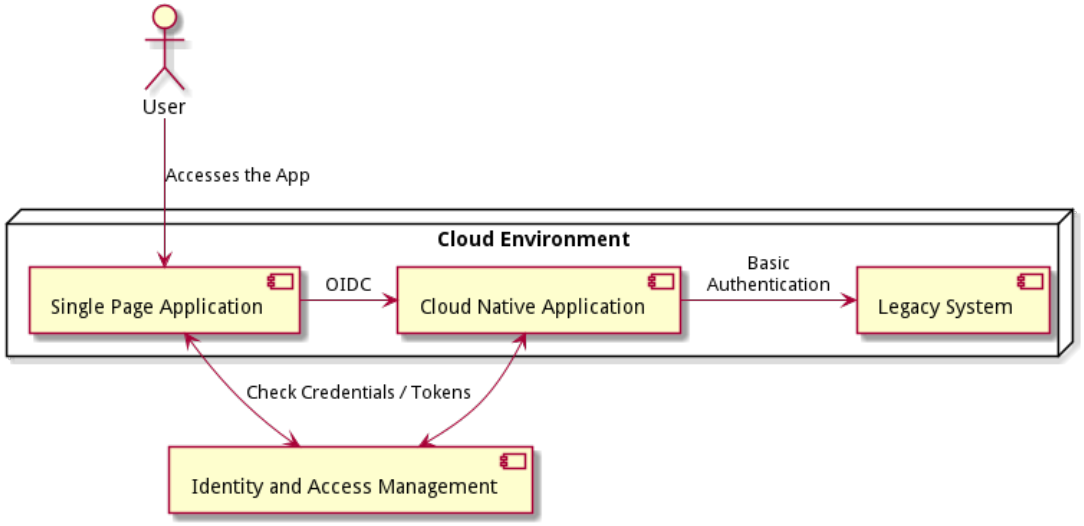

Figure 3.1: Microservice Architecture that contains modern applications as well as legacy services.

We consider the components in Figure 3.1:

- **User**: A person with access to the application
- **Single Page Application**: A modern single page application (SPA)
- **Identity and Access Management (IAM)**: Identity Provider for the solution (does not necessarily reside in the same cloud)
- **Cloud Native Application (CNA)**: A modern API application and primary access point for the client
- **Legacy System**: Legacy service that is called by service a to fetch some additional data

In the practice, we encountered the stated scenario at various points in time. Legacy services may not be the primary use case, another one is the usage of third-party applications without any access to the sourcecode.

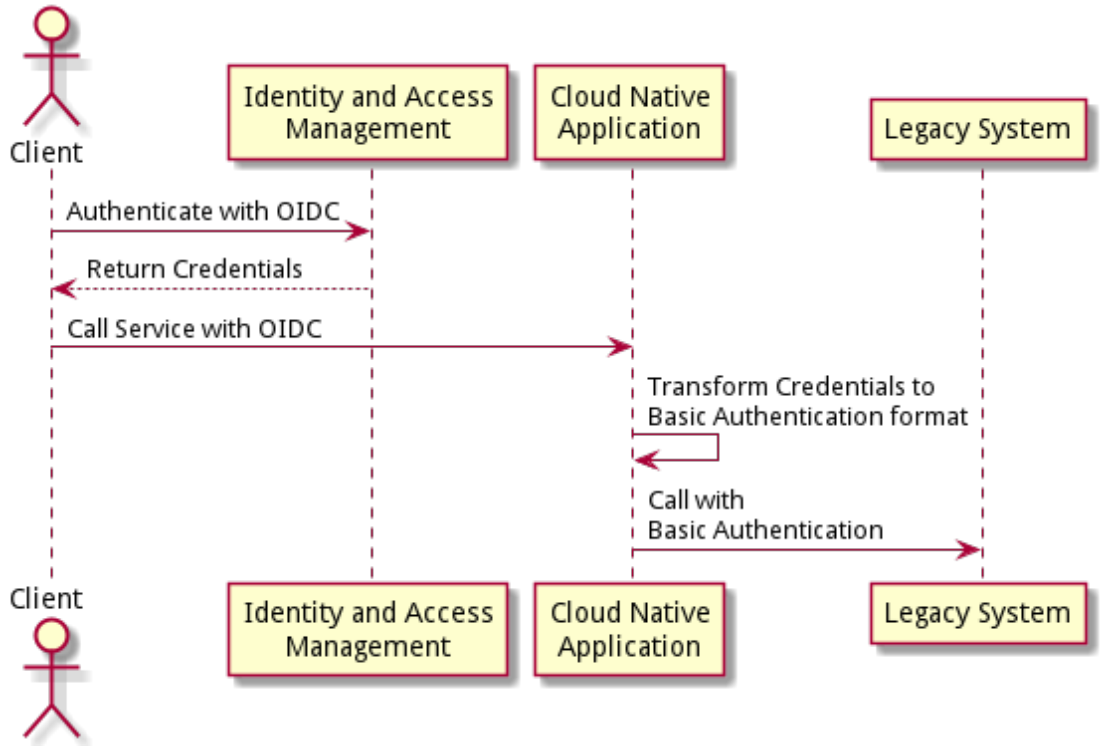

Figure 3.2: Current state of the art of accessing legacy systems from modern services with differing authentication schemes.

The invocation sequence in Figure 3.2 shows the process of communication in such a scenario. In Figure 3.2, the SPA authenticates against an arbitrary IAM. The CNA is the modern backend that supports the SPA as a backend API. Therefore, the CNA provides functionality for the SPA. The legacy application, for example an old ERP with order information, was moved into the cloud, but is not refactored nor rewritten to communicate with modern authentication technologies.

In this scenario, the SPA calls some API on the CNA that then will call the legacy system to get additional information to present to the user. Since the SPA and the CNA communicate with the same authentication technology, the call is straightforward. The SPA authenticates itself and obtains an access token. When calling the service (the CNA), the token is transmitted and the service can check with the IAM if the user is authorized to access the system. When the CNA then calls the legacy system for additional information, it is required to translate the user-provided credentials to a format that the legacy system understands. In the example, the legacy system is only able to handle Basic Authentication (RFC7617), as explained in Section 2.6.1. This means, if the

CNA wants to communicate with the legacy system, it must implement some translation logic to change the user credentials into the typical Basic Authentication Base64 encoded format of `<Username>:<Password>`. Hence, code changes are introduced to the CNA since the legacy system is not likely to be easily maintainable.

3.2 External Authentication and Identity Transport

In practice, no current solution exists that allows credentials to be transformed between authentication schemes. The service mesh "Istio" provides a mechanism to secure services that communicate with mTLS (mutual TLS) (Istio Authors 2021b) as well as an external mechanism to provide custom authentication and authorization capabilities (Istio Authors 2021a). The concept of Istio works well when all applications in the system share the same authentication scheme. As soon as two or more schemes are in place, the need for transformation arises again.

In fact, the external authentication feature of Istio is based on Envoy. Istio uses Envoy as an injected sidecar. Another prominent API gateway, "NGINX"[2], implements a similar external authentication mechanism (F5 Inc. Authors 2021). However, Envoy implements a more fine-grained control over the HTTP request. As an external authentication service for Envoy, the result may change HTTP headers in the request and the response. While both gateways offer a way of external authentication and authorization, they cannot transform the user-credentials on their own. In Envoy, the external service may attach or modify HTTP headers, while in NGINX, the external service may only allow or block a request.

There exist techniques, such as SAML (Security Assertion Markup Language) or JWT (JSON web token) profiles, to transmit an identity of a user to other services. However, SAML only describes the format of the identity itself, not the translation between varying credentials. SAML works when all participating services understand SAML as well. If a legacy system is not able to parse and understand SAML, the same problem arises.

All the discussed technologies and applications above do not support the dynamic conversion of user credentials. While Istio solves the communication and enables mTLS between services, it is not able to translate credentials between services. SAML gives a common format for an identity of a user, but it is an authentication scheme on its own and thus the "translation problem" still exists.

3.3 Missing Dynamic Credential Transformation

The situation described in the previous sections introduces several problems. It does not matter whether the legacy system is a third party application to which no code changes

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

can be applied to, or if it is an application that cannot be updated for the time being. Most likely, the code change to provide the ability to communicate will be introduced into the CNA. This adds the risk of errors since new code must be produced, which would not be necessary if the legacy service was refactored. Also, changing the CNA to communicate with the legacy software may be a feasible solution in a small setup. But as the landscape of the application grows, this solution does not scale well.

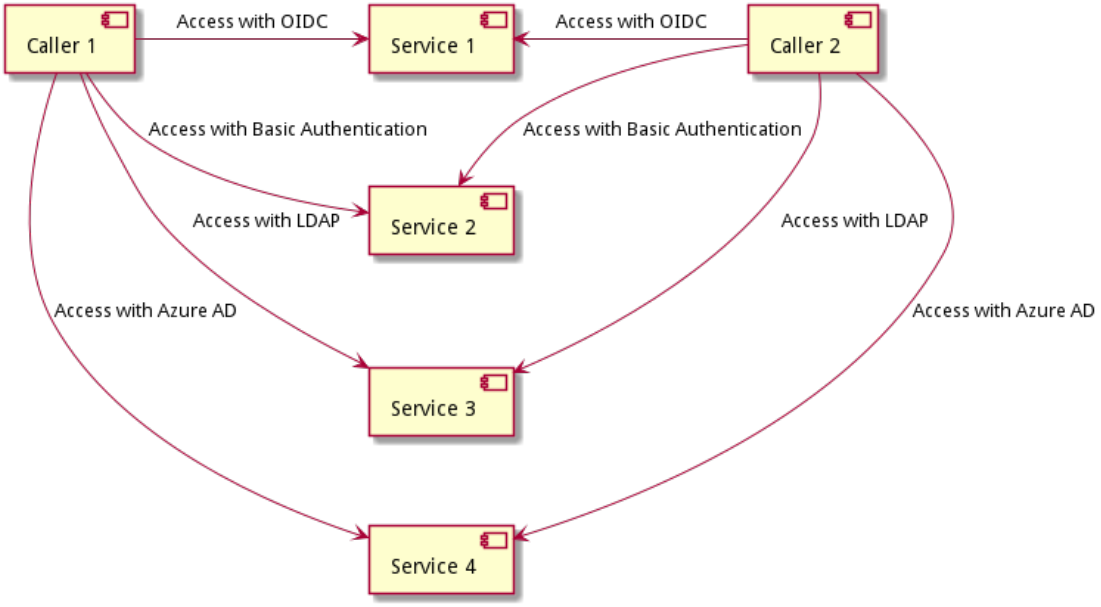

Figure 3.3: Matrix Problem in Service Landscape

The matrix problem, as depicted in Figure 3.3, shows that the number of conversion mechanisms increases with each service and each authentication method. As the landscape and the different methods of authentication grow, it is not a feasible solution to implement every authentication scheme in all the callers. In Figure 3.3, "Caller 1" is required to transform the user credentials into four different formats to communicate with services one to four. When another caller enters the landscape, it must implement the same four mechanisms as well.

Another issue that emerges with this transformation of credentials: The credentials leak into the trust zone. As long as each service is in the same trust zone (for example, in the same data-center in the same cluster behind the same API gateway), this may not be problematic. As soon as the communication is between data centers, the communication and the credentials must be protected. It is not possible to create a zero trust (assumption, that an attacker is always present, see Section 2.6.3) environment with the need of knowledge about the target's authentication schemes.

Figure 3.4 shows the problem when communicating between trust zones. While the two services in "Trust Zone A" may implicitly trust each other, because they are in the same

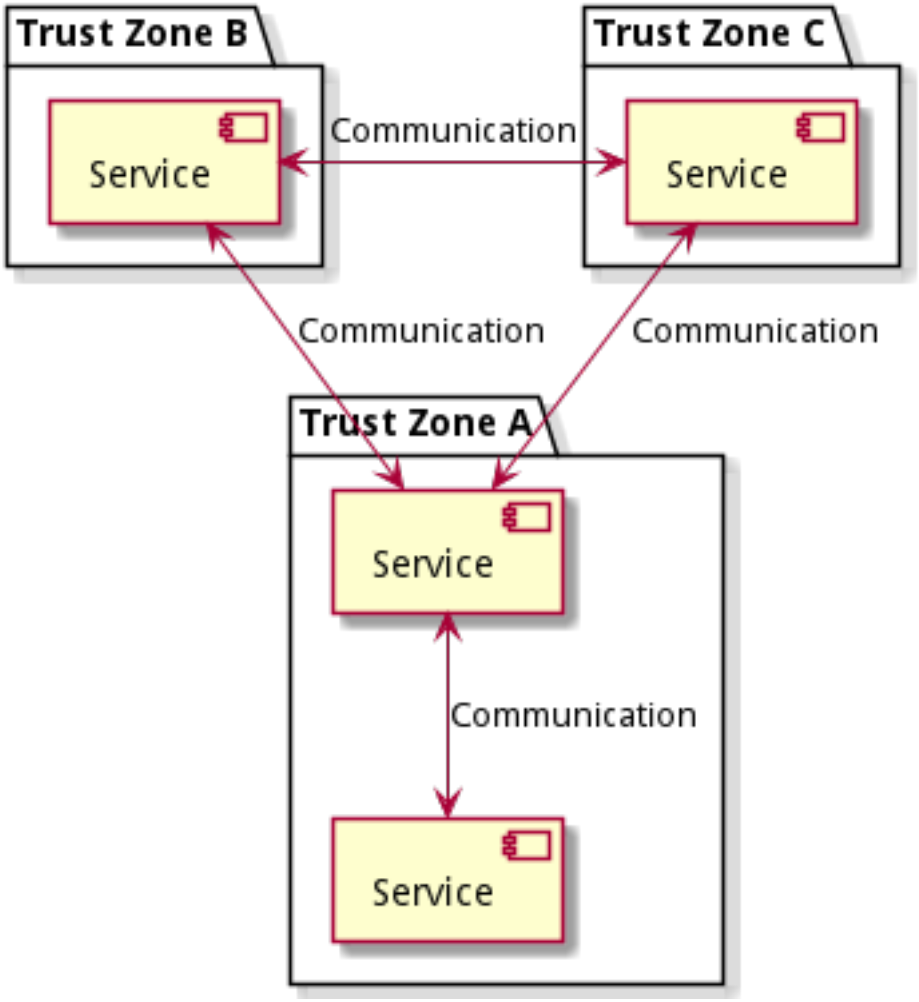

Figure 3.4: Zero Trust Environment

trust zone behind the same API gateway, the problem of leaking credentials is not critical. As soon as the three zones communicate with each other, credentials leave the trust zone and enter another zone. When implementing zero trust, each service in the system must verify the integrity and validity of the presented credentials again. There must not be any implicit trust based on a trust zone.

Service meshes may provide a way to secure communication between services, but they are not able to transform credentials to a required format for any legacy application yet. It would be a possible solution to enable service meshes to transform credentials. However, service meshes introduce another layer of complexity on top of the environment. Such a system should be easy to use.

Other technologies - such as credential vaults - provide a similar problem. The vault is the central weakness in the system. If the vault is attacked, the whole trust zone may fail. While a credential vault would provide a way to share credentials between services, it does not mitigate the need of transformation. A vault, like "Vault by HashiCorp"[3] typically provides a secure way to inject credentials into a system. The vaults do not transform credentials for the destination. However, such a vault could be used as secure storage of credentials for such a system that enables the described transformation. In the given example above, the secure vault could be used to store the static Basic Authentication credentials for the legacy service. The transformer can then access the vault to fetch the needed credentials for the target system.

[3] https://www.vaultproject.io/

4 Distributed Authentication Mesh

This section gives a general overview of the proposed solution. Furthermore, boundaries of the solution are provided along with common software engineering elements like requirements, non-functional requirements, and the documentation of the architecture.

The proposed architecture provides a generic description for a solution to the described problem. For this project, a proof of concept (POC) gives insights into the general topic of manipulating HTTP requests in-flight. The POC is implemented to run on a Kubernetes cluster to provide a practical example.

4.1 Definition

A solution for the stated problems in Section 3.3 must be able to transform arbitrary credentials into a format that the target service understands. For this purpose, the architecture contains a service that runs as a sidecar among the target service. This sidecar intercepts requests to the target and transforms the Authorization HTTP header. The sidecar is - like in a service mesh - used to intercept inbound and outbound traffic.

However, the solution **must not** interfere with the data flow itself. The problem of proxying data from point A to B is well solved. In the given work, an Envoy proxy is used to deliver data between the services. Envoy allows the usage of an external service to modify requests in-flight.

4.2 Goals and Non-Goals of the Project

This section presents the functional and non-functional requirements and goals for the solution. It is important to note that the implemented proof of concept (POC) will not achieve all goals. Further work is needed to implement a solution according to the architecture that adheres to the stated requirements.

In Table 4.1, we present the list of functional requirements or goals (REQ) for the proposed solution and the project in general.

Table 4.1: Functional Requirements

Name	Description
REQ 1	The translator module must be able to transform given credentials into the specified common language and the common format back into valid credentials.
REQ 2	The translator is injected as a sidecar into the solution. In Kubernetes this is done via an operator.
REQ 3	Beside the translator, an Envoy proxy is injected into the service in question to handle the data flow. This injection is performed by the operator as well.
REQ 4	Translators do only modify HTTP headers. They do not interfere with the data that is transmitted. Any information that needs to be forwarded must reside in the HTTP headers.
REQ 5	The automation engine can decide which elements are relevant for the authentication mesh.
REQ 6	The automation engine - if it exists - enhances objects with the proxy and translator engine.

In Table 4.2, we show the non-functional requirements or non-goals (NFR) for the proposed solution.

Table 4.2: Non-Functional Requirements

Name	Description
NFR 1	First and foremost, the solution **must not** be less secure than current solutions.
NFR 2	The solution must adhere to current best practices and security mechanisms. Furthermore, it **must** be implemented according to the standards of the practice to mitigate security issues as stated in the OWASP Top Ten (https://owasp.org/www-project-top-ten).
NFR 3	The concept of the solution is applicable to cluster orchestration software other than Kubernetes. The architecture provides a general way of solving the stated problem instead of giving a proprietary solution for one vendor. The concept should even be realizable for non-orchestration platforms like a Windows operating system.
NFR 4	The translation of the credentials should not extensively impact the timeframe of an arbitrary request. In production mode, the additional time to check and transform the credentials **should** not exceed 100ms. This is a general recommendation and some authentication mechanism may exceed the stated 100ms.

Name	Description
NFR 5	The solution is modular. It is extensible with additional "translators" that provide the means of transforming the given credentials to other target formats.
NFR 6	The solution may run with or without a service mesh. It is a goal that the solution can run without a service mesh to reduce the overall complexity, but if a service mesh is already in place, the solution must be able to work with the provided infrastructure.
NFR 7	The architecture **must** be scalable. The provided software must be able to scale according to the business needs of the overall system.
NFR 8	Each translator **should** only handle one authentication scheme to ensure separation of concerns and scalability of the whole solution.
NFR 9	The solution depends on an external software for data transmission. The solution **must not** interfere with the data plane. Error handling of the data plane is handled by the external application.
NFR 10	The solution handles errors in the translation and the automation engine according to the architectural description.

These goals and non-goals define the first list of REQ and NFR. During further work, this list may be changed to adjust to new challenges as the solution is implemented.

4.3 Differentiation from other Technologies

To distinguish this project from other technologies, this section gives a differentiation to two specific topics. The given topics stand for a general architectural idea and the contrast to the presented solution.

4.3.1 Security Assertion Markup Language

The "Security Assertion Markup Language" (SAML) is a so-called "Federated Identity Management" (FIdM) standard. SAML, OAuth, and OIDC represent the three most popular FIdM standards. SAML is an XML framework for transmitting user data, such as authentication, entitlement, and other attributes, between services and organizations (Naik and Jenkins 2017).

While SAML is a partial solution for the stated problem, it does not cover the use case when credentials need to be transformed to communicate with a legacy system. SAML enables services to share identities in a trustful way, but all communicating partners must implement the SAML protocol to be part of the network. This project addresses the specific transformation of credentials into a format for some legacy systems. The basic

idea of SAML may be used as a baseline of security and the general idea of processing identities.

4.3.2 WS-*

The term "WS-*" contains a broad class of specifications within the "Web Services Description Language" (WSDL) and "Simple Object Access Protocol" (SOAP) context. The specifications were created by the World Wide Web Consortium (W3C). However, the consortium never finished and released the specification.

SOAP is a protocol to exchange information between services in an XML encoded message. It provides a way of communication between web services. A SOAP message consists of an "envelope" that contains a "body" and an optional "header" to transfer encoded objects (Curbera et al. 2002). An example SOAP message from Curbera et al. (2002):

```
POST /travelservice
SOAPAction: "http://www.acme-travel.com/checkin"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <et:eTicket xmlns:et="http://www.acme-travel.com/eticket/schema">
      <et:passengerName first="Joe" last="Smith"/>
      <et:flightInfo
        airlineName="AA"
        flightNumber="1111"
        departureDate="2002-01-01"
        departureTime="1905"/>
    </et:eTicket>
  </SOAP:Body>
</SOAP:Envelope>
```

WSDL is an XML-based description of a web service. The goal of WSDL is to provide a description of methods that may be called on a web service. WSDL fills the needed endpoint description that SOAP is missing. While SOAP provides basic communication, WSDL defines the exact methods that can be called on an endpoint (Curbera et al. 2002).

The distributed authentication mesh differs from WS-* such that there is no exact specification required for the target service. While the solution contains a common domain language - a SOAP-like protocol to encode data - it does not specify the endpoints of a service. The solution merely interacts with the HTTP request that targets a specific service and transforms the credentials from the common format to the specific format.

Of course, certain authentication schemes need additional information to generate their user credentials out of the data.

4.4 Use Case of Dynamic Credential Transformation

The usefulness of such a solution shows when "older" or monolithic software moves to the cloud or when third-party software is used that provides no accessible source code.

Communicate with legacy software

Precondition: Cloud-Native Application (CNA) and legacy software is deployed with their respective manifests and the sidecars of the mesh are running.

1. The user is authenticated against the CNA
2. The user tries to access a resource on the legacy software
3. The CNA creates a request and "forwards" the credentials of the user
4. The proxy intercepts the request and forwards the credentials to the transformer
5. The transformer verifies the credentials and transforms them into a domain-specific format
6. The proxy replaces the headers and forwards the request
7. The receiving proxy forwards the domain-specific format to the translator of the target
8. The translator casts the credentials into the specific authentication scheme credentials
9. The receiving proxy forwards the request to the target service with the updated HTTP headers

Postcondition: The communication has taken place and no credentials have left the source service (the CNA). Furthermore, the legacy service does not know what specific authentication scheme was used by the source to identify the user.

This use case can be changed such that the receiving service is not a legacy software but some third-party application where the source code is not accessible.

4.5 Architecture of the Solution

The following sections provide an architectural overview over the proposed solution. The brief description gives an initial overview of the architecture and the idea. Afterwards, an abstract architecture describes the concepts behind the distributed authentication mesh. Then the architecture is concretized with platform-specific examples based on Kubernetes.

The reader should note that the proposed architecture does not match the implementation of the POC to the full extent. The goal of this project is to provide an abstract idea to implement such an authentication mesh, while the POC proves the ability of modifying HTTP requests in-flight.

4.5.1 Brief Description

In general, when some service wants to communicate with another service and the user does not need to authenticate himself, most likely a federated identity is in use. This means that at some point, the user validates his own identity and is then authenticated in the whole zone of trust. This does not contradict a zero-trust environment. A federated identity can be validated by each service and thus may be used in a zero-trust environment.

To achieve such a federated identity with diverging authentication schemes, the solution converts validated credentials (like access tokens) to a domain specific language (DSL). This format, in conjunction with a proof of the sender, validates the identity over the wire in the communication between services without the need of additional authentication. When all parties of a communication are trusted through verification, no information about the effective credentials leaks into the communication between services.

The basic idea of the distributed authentication mesh is to replace any user credentials from an outgoing HTTP request with the DSL representation of the users identity. On the receiving side, the DSL encoded identity in the incoming HTTP request is transformed to the valid user credentials for the target service.

Since the topic of the mesh is about security, error handling is a delicate matter. The mesh does depend on existing infrastructure and principles. Thus, error handling is limited to the translator engine. When the translator encounters any error, the request is denied.

4.5.2 Abstract and Conceptional Architecture

This section describes the architecture of the proposed solution in an abstract and generalized way. The concepts are not bound to any specific platform or a specific implementation nor required to run in a cloud environment. The concepts could be implemented as a “fat-client” solution for a Windows machine as well.

Figure 4.1 shows the abstract solution architecture. In the “support” package, generally available elements provide utility functions to the mesh. The solution requires a public key infrastructure (PKI) to deliver key material for signing and validation purposes. This key material may also be used to secure the communication between the nodes (or applications). Furthermore, configuration and secret storage enable the applications to store and retrieve configurations and secret elements like passwords or key material.

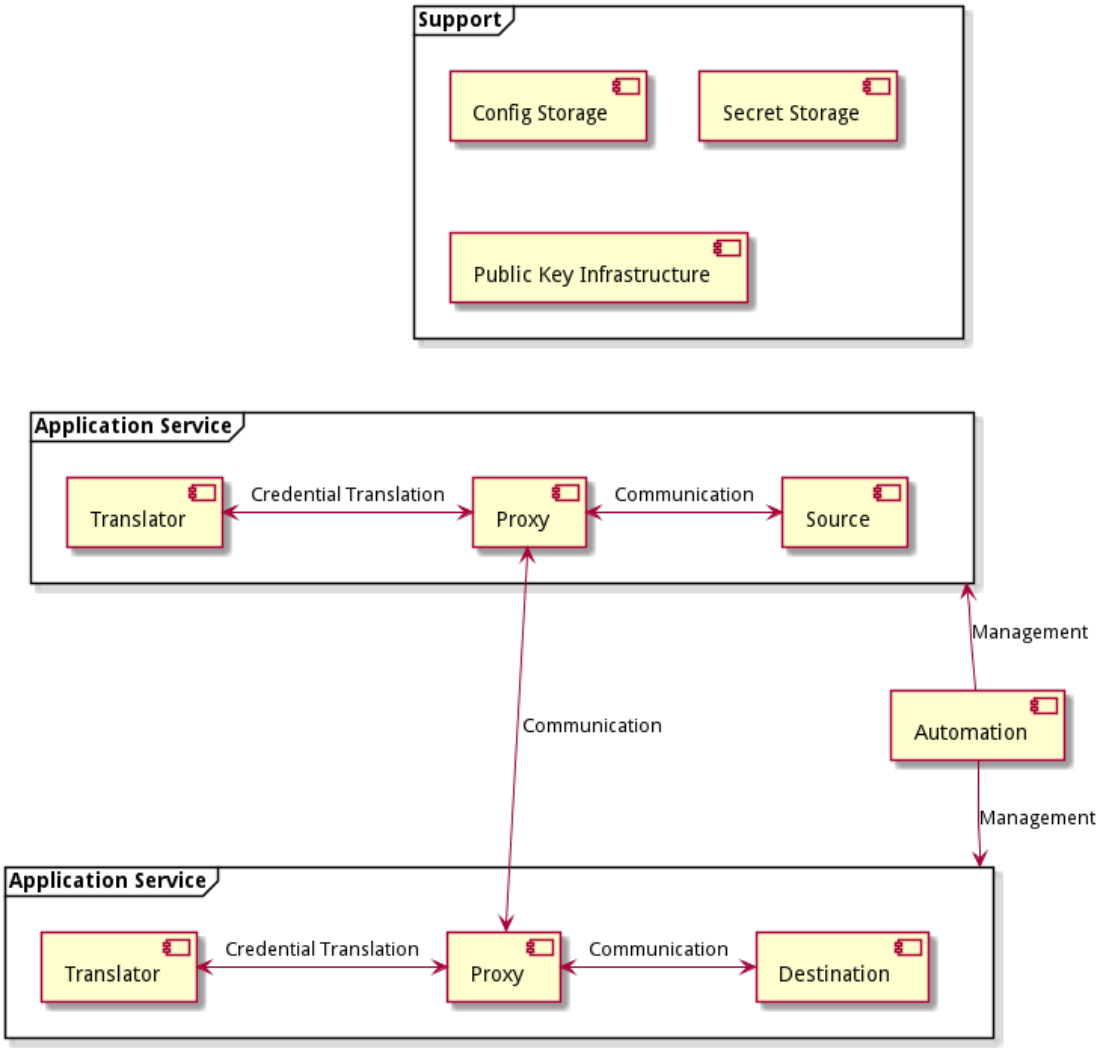

Figure 4.1: Abstract Solution Architecture

Additionally, an optional automation component watches and manages applications. This component enhances the application services with the required components to participate in the distributed authentication mesh. Such a component is strongly suggested when the solution is used in a cloud environment to enable dynamic usage of the mesh. The automation injects the proxies, translators, and the required configurations for the managed components.

A (managed) application service consists of three parts. First, the source (or destination) service, which represents the deployed application itself, a translator that manages the transformation between the DSL of the identity and the implementation specific authentication format and a proxy that manages the communication from and to the application.

The communication between instances in the authentication mesh is handled by the proxies. The mesh must not interfere with the data transmission, it is only responsible to transform HTTP headers. Handling errors on the data plane is not part of the mesh and must be done by the implementation of the proxy.

4.5.3 Platform-Specific Example in Kubernetes

For these sections, the architecture shows elements of a Kubernetes cloud environment. The reason is to describe the specific architecture in a practical way. However, the general idea of the solution may be deployed in various environments and is not bound to a cloud infrastructure. Table 2.1 gives an overview of used terms and concepts in Kubernetes which are used to describe the platform-specific architecture.

Since the example is Kubernetes specific, error handling and recovery mechanisms of Kubernetes can be used. So if a part of the mesh dies due to an unexpected error, Kubernetes is responsible to restart the part. Furthermore, Kubernetes is the orchestrator which takes actions to provide the running state of all applications. If any errors are encountered, propper logging must be provided.

4.5.3.1 Automation with an Operator

In case of a Kubernetes infrastructure, the automation part is done by an operator pattern as explained in Section 2.3. The automation part of the mesh is optional. When no automation is provided, the required proxy and translator elements must be started and maintained by some other means. However, in the context of Kubernetes, an operator pattern enables an automated enhancement and management of applications.

The operator in Figure 4.2 watches the Kubernetes API for changes. When deployments or services are created, the operator enhances the respective elements. “Enhancing” means that additional pods are injected into a deployment as sidecars. The additional pods consist of the proxy and the translator. While the proxy manages incoming and

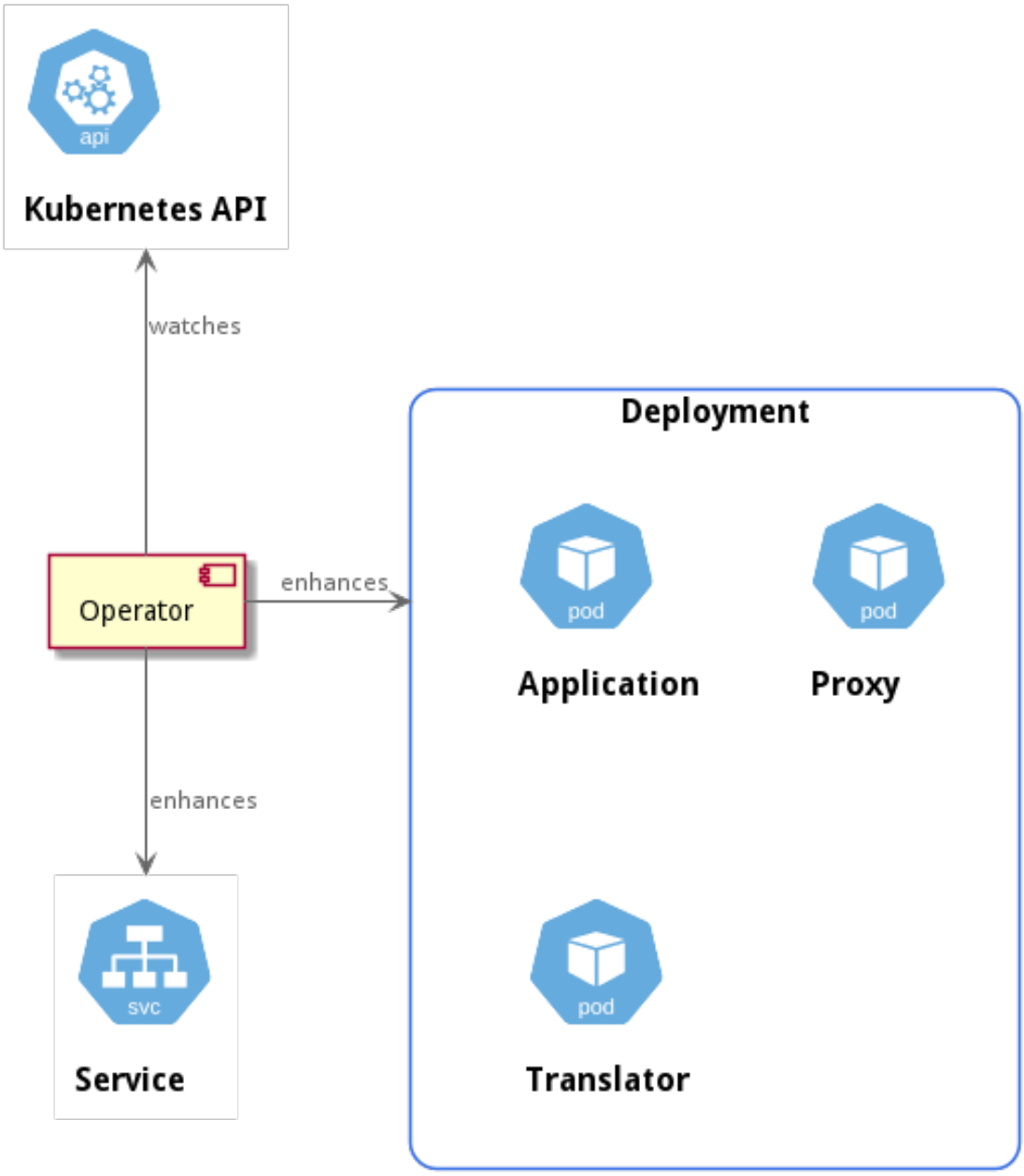

Figure 4.2: Automation with an Operator in a Kubernetes Environment

outgoing communication, the translator manages the transformation of credentials from and to the DSL.

To determine if an object is relevant for the automation, the operator uses the logic shown in Figure 4.3. If the object in question is not a deployment (or any other deployable resource, like a "Stateful Set" or "Daemon Set") or a service, then it is not relevant for the mesh. If the object is not configured to be part of the mesh, then the automation ends here as well. The last step is to inject or reconfigure elements of the object depending on its effective type.

The sequence that enhances deployments and services is shown in Figure 4.4. The operator registers a "watcher" for deployments and services with the Kubernetes API. Whenever a deployment or a service is created or modified, the operator receives a notification. Then, the operator checks if the object in question "is relevant" by checking if it should be part of the authentication mesh. This participation can be configured - in the example of Kubernetes - via annotations, labels or any other means of configuration. If the object is relevant, depending on the type, the operator injects sidecars into the deployment or reconfigures the service to use the injected proxy as targeting port for the network communication.

If the automation engine encounters errors, it relies on Kubernetes to perform actions to reach a meaningful state. Since the engine runs on Kubernetes, if any operational errors occur, the application is restarted by Kubernetes. Logging is essential to find such errors. If deployments and services cannot be modified, the operator shall try it again in the next reconciliation-cycle.

4.5.3.2 Public Key Infrastructure

The role of the public key infrastructure (PKI) in the solution is to build the trust anchor in the system.

Figure 4.5 depicts the relation of the translators and the PKI. When a translator starts, it acquires trusted key material from the PKI (for example with a certificate signing request). This key material provides the possibility to sign the identity that is transmitted to the receiving party. The receiving translator can validate the signature of the identity and the sending party. The proxies are responsible for the communication between the instances.

The sequence in Figure 4.6 shows how the PKI is used by the translator to create key material for itself. When a translator starts, it checks if it already generated a private key and obtains the key (either by creating a new one or fetching the existing one). Then, a certificate signing request (CSR) is sent to the PKI. The PKI will then create a certificate with the CSR and return the signed certificate. The provided sequence shows one possible use case for the PKI. During future work, the PKI may also be used to secure communication between proxies with mTLS.

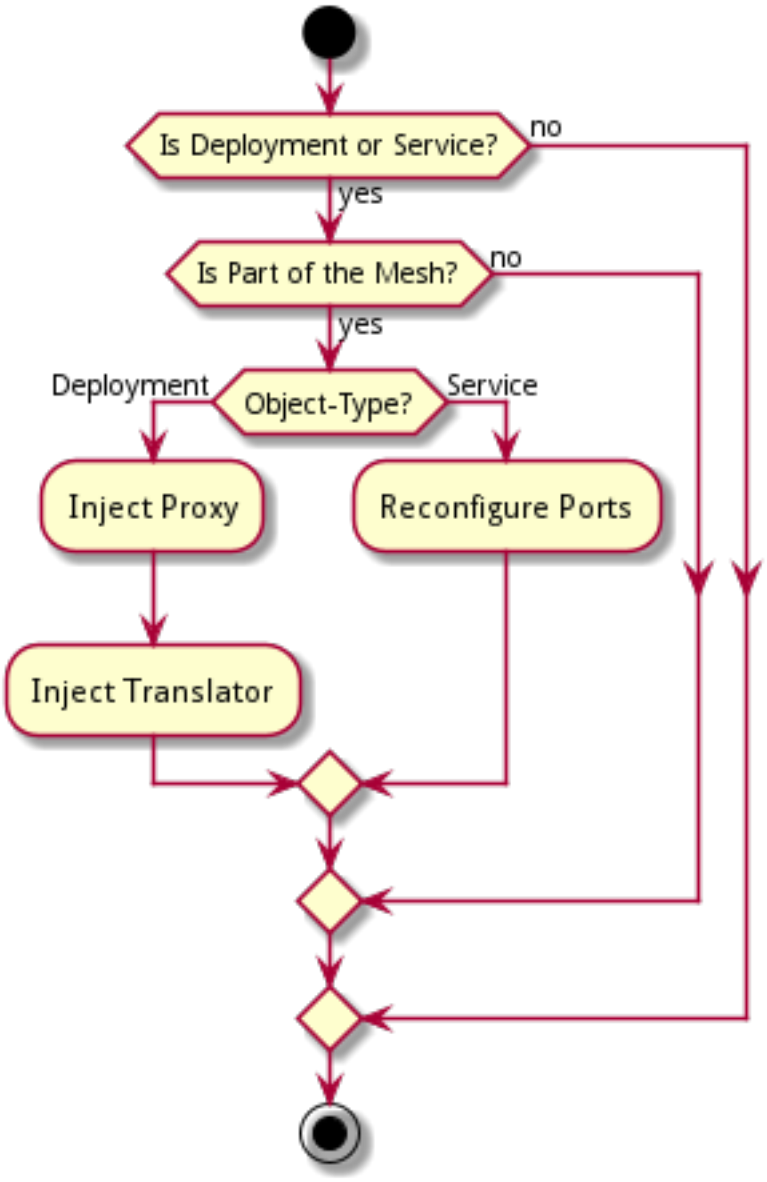

Figure 4.3: Determination of the Relevance of a Deployment or a Service

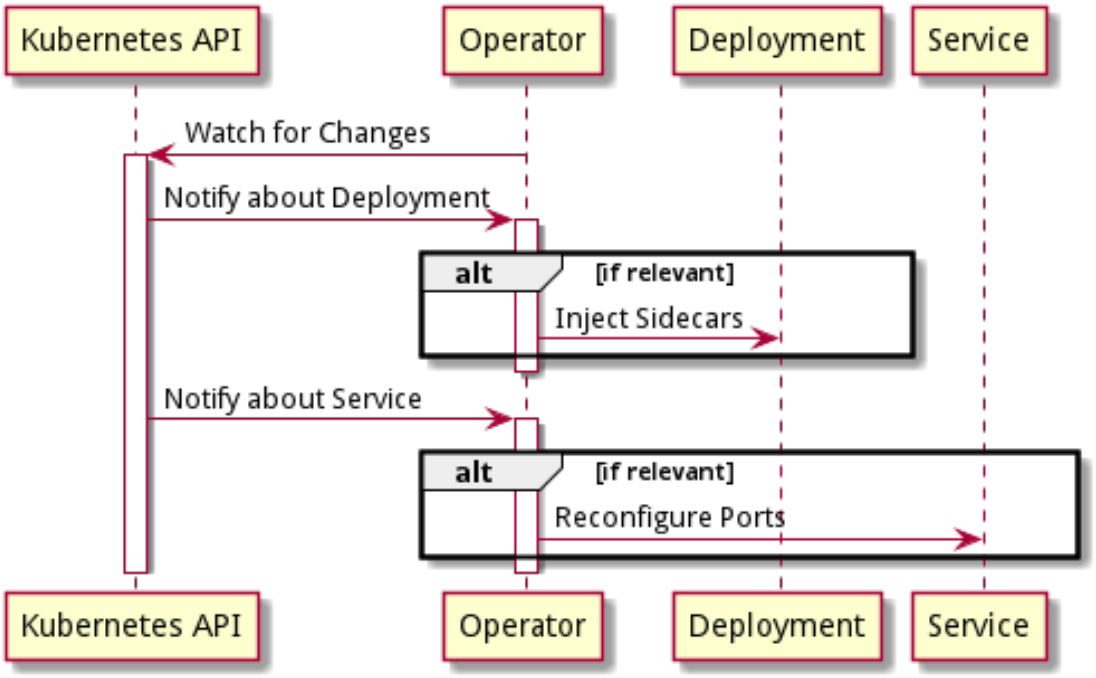

Figure 4.4: Automated Enhancement of a Deployment and a Service

When communication happens, the proxy forwards the HTTP headers, that contain the transfered identity of the user in the DSL, to the translator. In case of a JWT token, the transformer may now confirm the signature of the JWT token with the obtained certificate since it is signed by the same Certificate Authority (CA). Then the transformation is performed and the proxy forwards the communication to the destination.

To increase the security and mitigate the problem of leaking certificates, it is adviced to create short living certificates in the PKI and refresh certificates periodically.

If the PKI encounters illegal signing requests, it must deny them. If any other unexpected errors happen, the application should log the error and then crashes to enable Kubernetes to restart the application again.

4.5.3.3 Networking with a Proxy

Networking in the proposed solution works with a combination of routing and communication proxying. The general purpose of the networking element is to manage data transport between instances of the authentication mesh and route the traffic to the source/destination.

As seen in Figure 4.7 the proxy is the mediator between source and destination of a communication. Furthermore, the proxy manages the translation of the credentials by

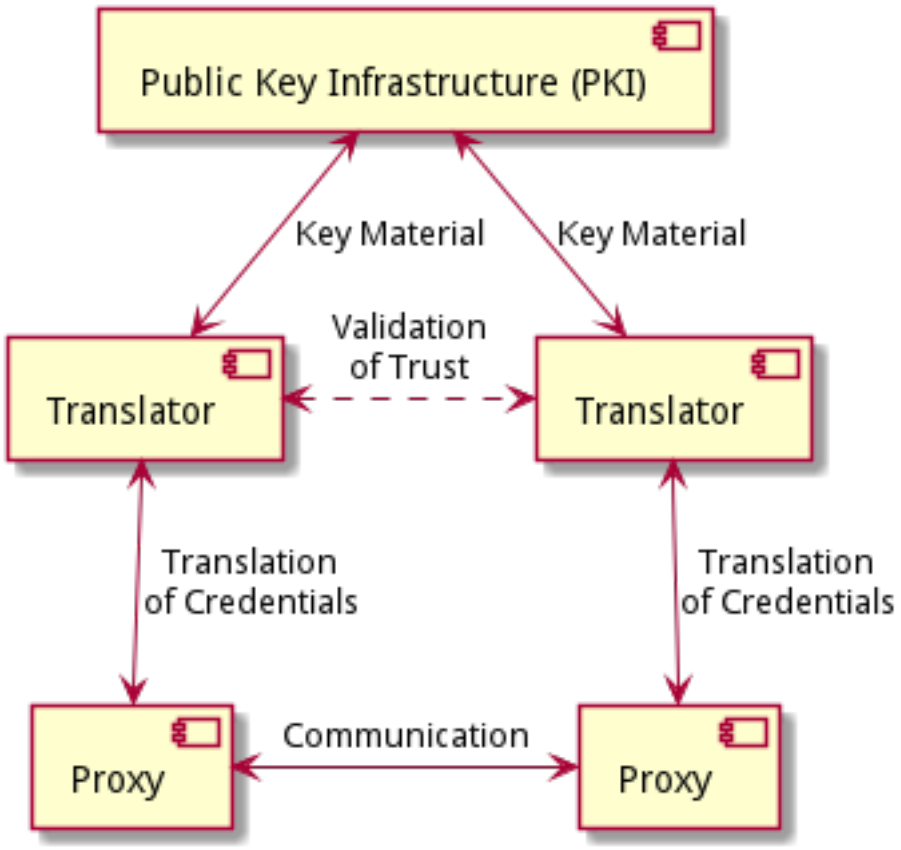

Figure 4.5: The Relation of the Public Key Infrastructure and the System

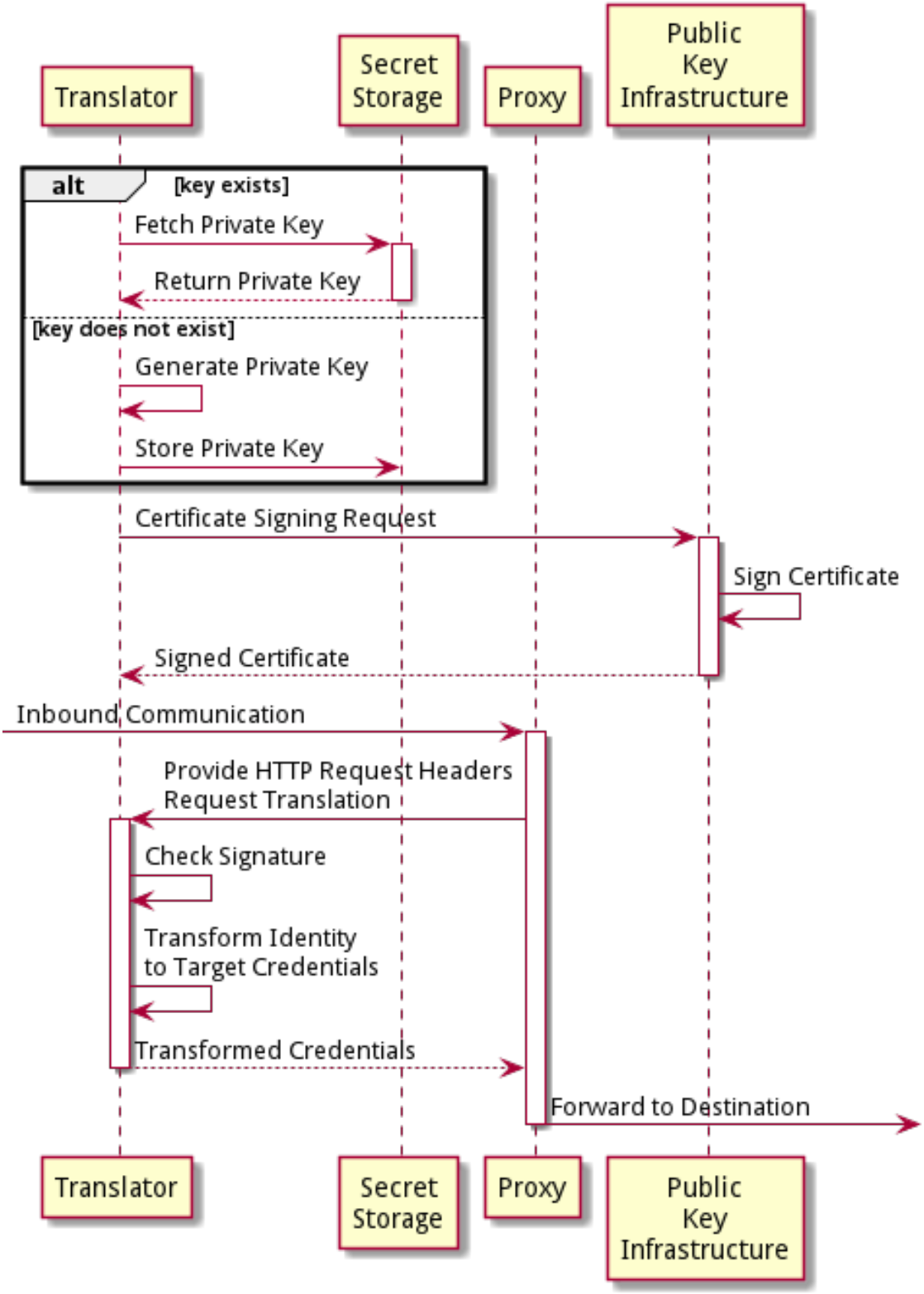

Figure 4.6: Provide Key Material to the Translator

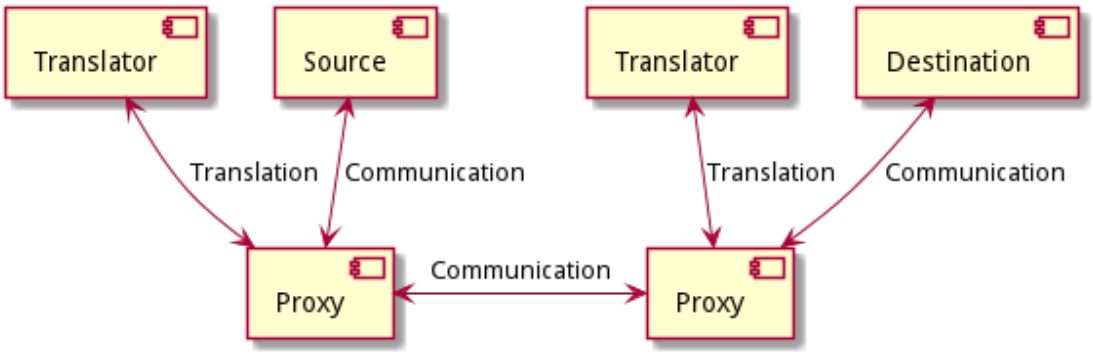

Figure 4.7: Networking with an Proxy

communicating with the translator to transform the identity of the authenticated user and transmit it to the destination where it gets transformed again. Additionally, with the help of the PKI, the proxy can verify the identity of the sender via mTLS.

Since the authentication mesh relies on external software to take care of communication and networking, error handling is off-loaded to that specific software as well. The authentication mesh does not guarantee any connectivity between parties of the mesh. In the specific example, if the configuration provided by the automation engine is faulty, Envoy will crash and log this matter to the output. Any other errors encountered by Envoy result in their respective HTTP error messages.

4.5.3.3.1 Inbound accepted Communication for an Application Figure 4.8 shows the general invocation during inbound request processing. When the proxy receives a request (in the given example by the configured Kubernetes service), it calls the translator with the HTTP request detail. The POC is implemented with an "Envoy" proxy. Envoy allows an external service to perform "external authorization"[1] during which the external service may:

- Add new headers before reaching the destination
- Overwrite headers before reaching the destination
- Remove headers before reaching the destination
- Add new headers before returning the result to the caller
- Overwrite headers before returning the result to the caller

The translator uses this concept to consume a specific and well-known header to read the identity of the authorized user in the DSL. The identity is then validated and transformed to the authentication credentials needed by the destination. Then, the translator instructs Envoy to set the credentials for the upstream. In the POC, this is done by setting the `Authorization` header to static Basic Authentication (RFC7617) credentials.

[1] https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/ext_authz_filter

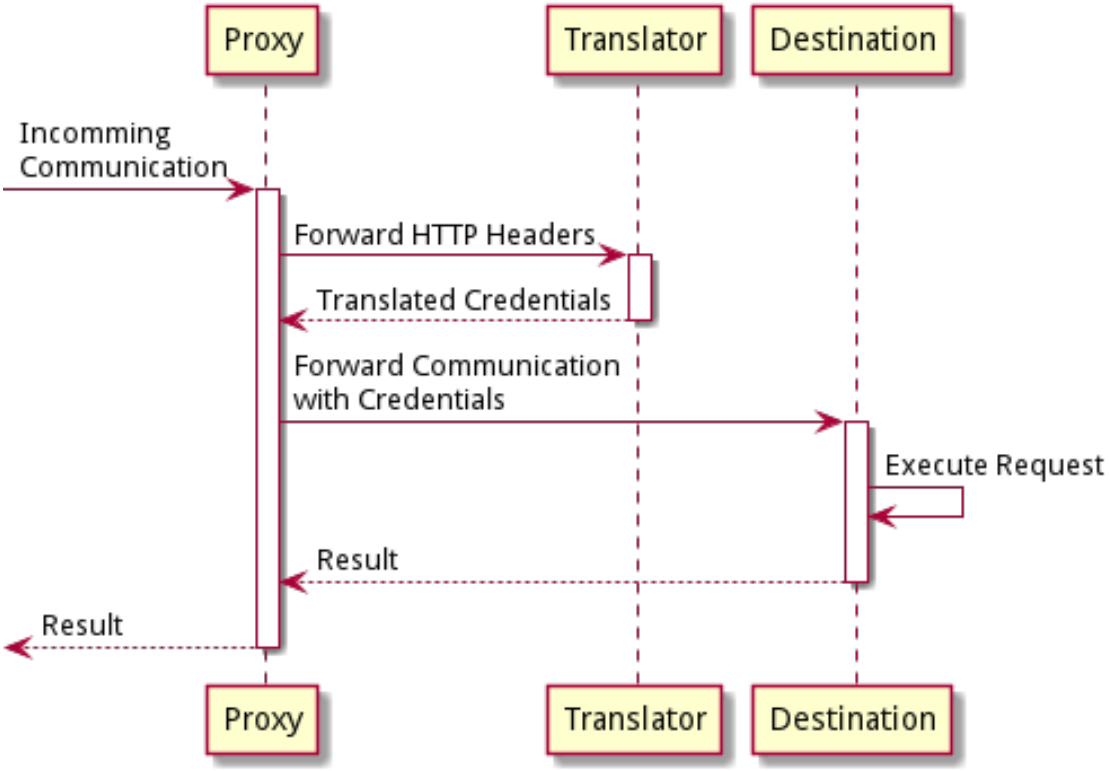

Figure 4.8: Inbound Accepted Networking Sequence

4.5.3.3.2 Inbound rejected Communication for an Application If the incoming communication contains faulty, invalid or no identification data, the proxy blocks the communication.

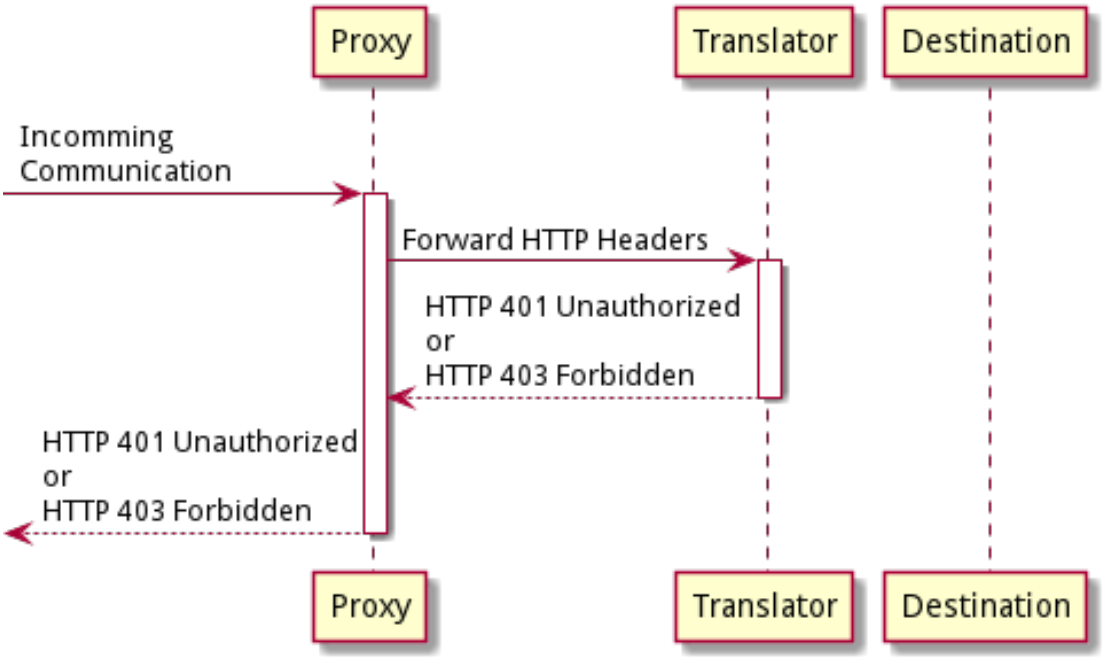

Figure 4.9: Inbound Rejected Networking Sequence

Figure 4.9 shows the sequence when no or invalid identity data is provided. The responses of the translator are defined in **RFC1945** and are the HTTP response status codes (Nielsen, Fielding, and Berners-Lee 1996). The translator distinguishes two cases:

- No identity data
- Invalid identity data

If no identity data is present, the translator will return a `HTTP 401 Unauthorized` error that is used when no authorization credentials are provided. When invalid authorization credentials are provided (a false or a modified identity) the translator will return `HTTP 403 Forbidden` which is used when credentials are provided, but they are not valid (Nielsen, Fielding, and Berners-Lee 1996, sec. 9.4).

4.5.3.3.3 Outbound Communication for an Application In Figure 4.10 the outbound traffic flow is shown. The proxy is required to catch all outbound traffic from the source and performs the reversed process of Figure 4.8 by transforming the provided information from the source to generate the common format with the users identity. This identity is then inserted into the HTTP headers and sent to the destination. At the sink, the process of Figure 4.8 takes place - if the sink is part of the authentication mesh.

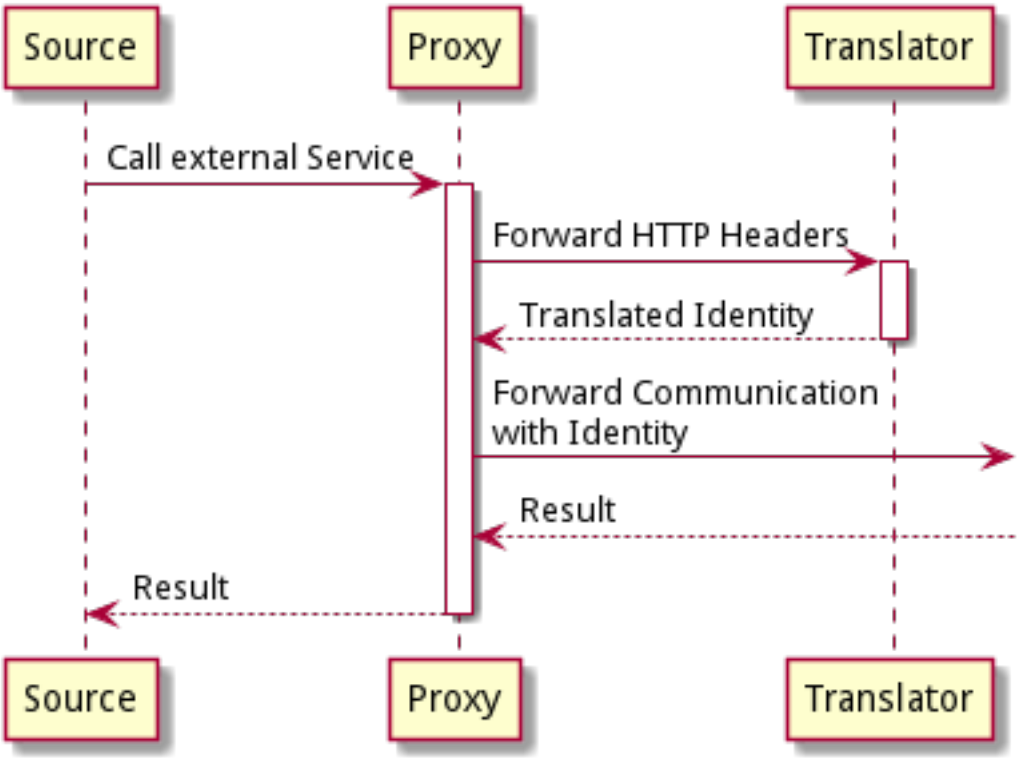

Figure 4.10: Outbound Networking Sequence

4.5.3.4 The Translation of Credentials to an Identity

The translator is responsible for transforming the identity from and to the domain-specific language. In conjunction with the PKI, the translator can verify the validity and integrity of the incoming identity.

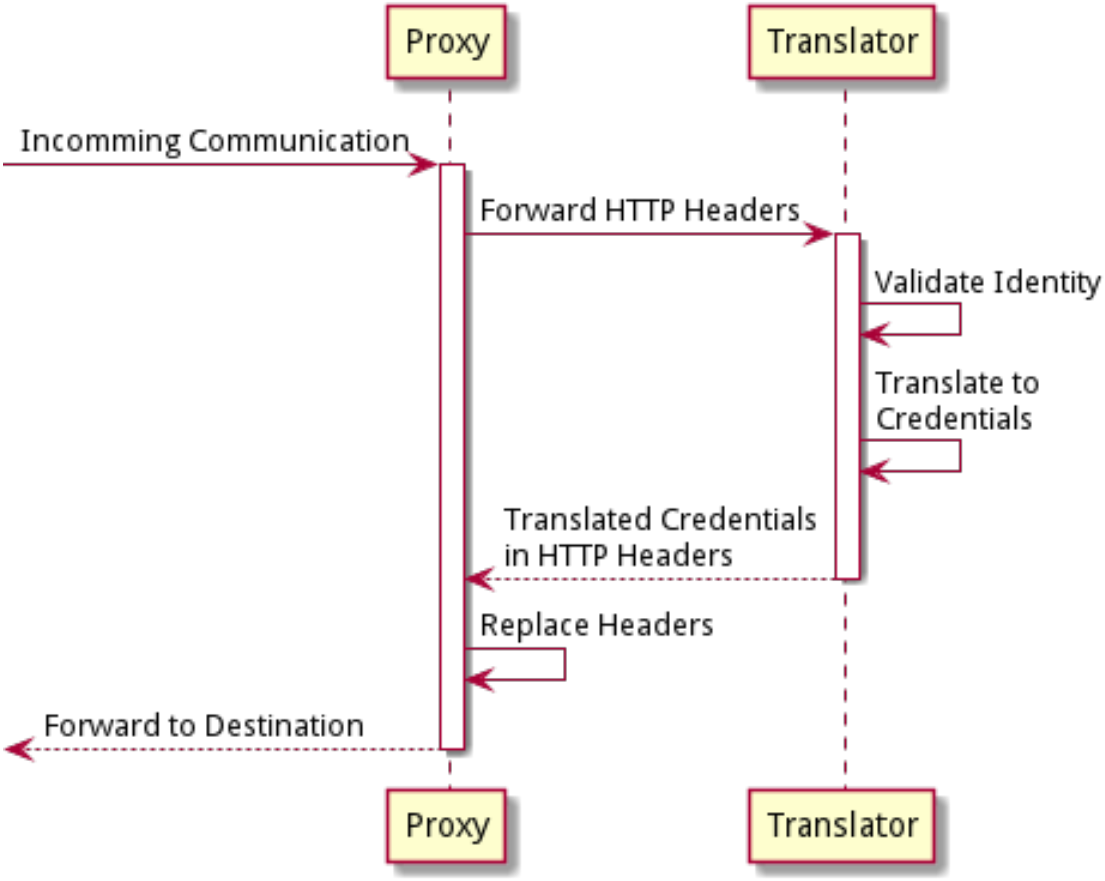

Figure 4.11: Translator Process

When the translator receives a request to create the required credentials, it performs the sequence of actions as stated in Figure 4.11. First, the proxy will forward the HTTP request data to the translator. Afterward, the translator checks if the transported identity is valid and signed by an authorized party in the authentication mesh. When the credentials are valid, they are translated according to the implementation of the translator. The proxy is then instructed with the actions to replace the transported identity with the correct credentials to access the destination.

The translator is the critical part of the authentication mesh. If it receives invalid credentials (for example an identity that has been tampered with), it must reject the request with a `HTTP 403 Forbidden` response. If no identity is provided at all, a `HTTP`

`401 Unauthorized` must be sent. When the translation engine encounters any unexpected error during translation of the identity (like not being able to access the secret storage, or failure of some database), it must reject the request. The translator must reject any request that cannot be transformed successfully.

In the POC, the proof of integrity is not implemented, but the transformation takes place, where a "Bearer Token"[2] is used to check if the user may access the destination and then replaces the token with static Basic Authentication credentials.

4.6 Securing the Communication between Applications

The communication between the proxies must be secured. Furthermore, the identity that is transformed over the wire must be tamper-proof. Two established formats would suffice, "SAML" and "JWT Tokens." While both contain the possibility to hash their contents and thus secure them against modification, JWT tokens are better designed for HTTP headers, since in current OIDC environments, JWT tokens are already used as access and/or identity tokens. JWT provides a secure environment with public and private claim names (Jones, Bradley, and Sakimura 2015, sec. 4.2, sec. 4.3).

Other options to encode the identity:

- Simple JSON
- YAML
- XML
- X509 Certificates
- Concise Binary Object Representation (CBOR)
- Any other structured format

The problem with other structured formats is that tamper protection and encoding must be done manually. JWT tokens provide a specified way of attaching a hashed version of the whole content and therefore provide a method of validating a JWT token if it is still valid and if the sender is trusted (Jones, Bradley, and Sakimura 2015). If the receiving end has his key material from the same PKI (and therefore the same CA), it can check the certificate and the integrity of the JWT token. If the signature is correct, the JWT token has been issued by a trusted and registered instance of the authentication network.

X509 certificates - as defined in **RFC5280** (Cooper et al. 2008) - introduce another valid way of transporting data and attributes to another party. "Certificate Extensions" can be defined by "private communities" and are attached to the certificate itself (Cooper et al. 2008, sec. 4.2, sec. 4.2.2).

[2] access token of an IDP.

While X509 certificates could be used instead of JWT to transport this data, using certificates would enforce the translator to act as intermediate CA and create new certificates for each request. From our experience, creating, extracting, and manipulating certificates, for example in C#, is not a task done easily. Since this solution should be as easy to use as it can be, manipulating certificates in translators does not seem to be a feasible option. For the sake of simplicity and the well-known usage, further work on this project will probably use JWT tokens to transmit the identity data.

4.7 Implementation Proof of Concept (POC)

To proof that the general idea of the solution is possible, a POC is implemented during the work of this project. The following technologies and environments build the foundation of the POC:

- Environment: The POC is implemented on a Kubernetes environment to enable automation and easy deployment for testing
- "Automation": A Kubernetes operator, written in .NET (C#) with the "Dotnet Operator SDK"[3]
- "Proxy": Envoy proxy which gets the required configuration injected as Kubernetes ConfigMap file
- "Translator": A .NET (F#) application that uses the Envoy gRPC definitions to react to Envoy's requests and poses as the external service for the external authorization
- "Sample Application": A solution of three applications that pose as demo case with:
  - "Frontend": An ASP.NET static site application that authenticates itself against "ZITADEL"[4]
  - "Modern Service": An ASP.NET API application that can verify an OIDC token from ZITADEL
  - "Legacy Service": A "legacy" ASP.NET API application that is only able to verify `Basic Auth` (RFC7617, see Section 2.6.1)

The POC addresses the following questions:

- Is it possible to intercept HTTP requests to an arbitrary service
- Is it further possible to modify the HTTP headers of the request
- Can a sidecar service transform given credentials from one format to another
- Can a custom operator inject the following elements:
  - The correct configuration for Envoy to use external authentication
  - The translator module to transform the credentials

[3] https://github.com/buehler/dotnet-operator-sdk
[4] https://zitadel.ch

Based on the results of the POC, the following further work may be realized:

- Specify the concrete format to transport identities
- Implement a secure way of transporting identities with validation of integrity
- Provide a production-ready solution for some translators and the operator
- Integrate the solution with a service mesh
- Provide a production-ready documentation of the solution
- Further investigate the possibility of hardening the communication between services (e.g. with mTLS)

For the solution to be production-ready, at least the secure communication channel between elements of the mesh as well as the DSL for the identity must be implemented. To be used in current cloud environments, an implementation in Kubernetes can provide insights on how to develop the solution for other orchestrators than Kubernetes.

4.7.1 Case Study for the POC

The demo application shows the need and the particular use case of the distributed authentication mesh. The application resides in an open-source repository on GitHub (https://github.com/WirePact/poc-showcase-app).

When installed in a Kubernetes cluster, the user can open (depending on the local configuration) the URL to the frontend application[5].

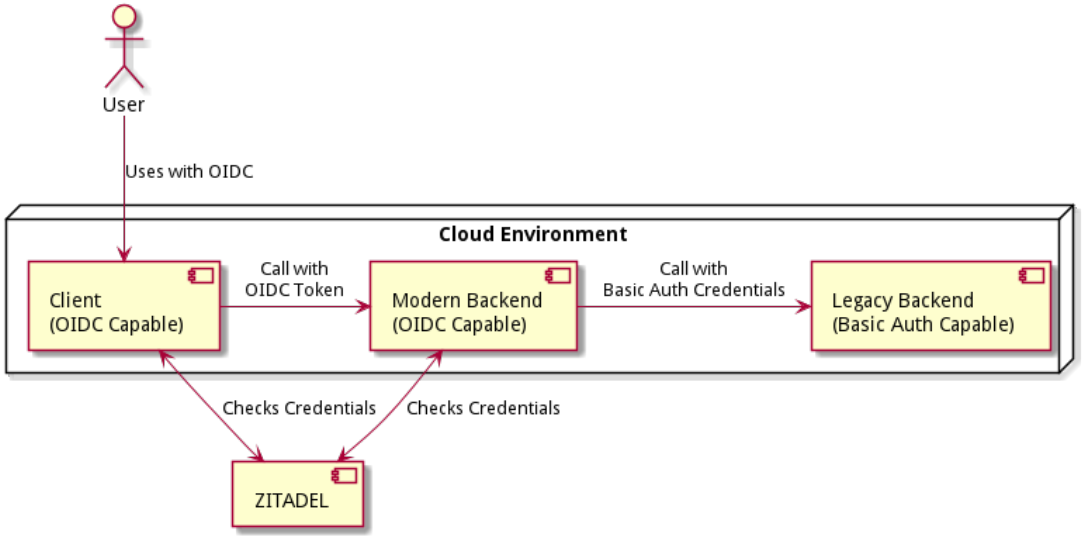

Figure 4.12: Component Diagram of the Case Study

[5]In the example, it is "https://kubernetes.docker.internal" since this is the local configured URL for "Docker Desktop"

Figure 4.12 gives an overview over the components in the showcase application. The system contains an ASP.NET Razor Page[6] application as the frontend, an ASP.NET API application with configured ZITADEL OIDC authentication as "modern" backend service, and another ASP.NET API application that only supports Basic Authentication as "legacy" backend. The frontend can only communicate with the modern API and the modern API is able to call an additional service on the legacy API.

In Figure 4.13, we show the process of a user call in the demo application. The user opens the web application and authenticates himself with ZITADEL. After that, the user is presented with the application and can click the "Call API" button. The frontend application calls the modern backend API with the access token from ZITADEL and asks for customer and order data. The customer data is present on the modern API so it is directly returned. To query the order data, the modern service relies on a legacy application which is only capable of Basic Authentication.

Depending on the configuration (i.e. the environment variable `USE_WIREPACT`), the modern service will call the legacy application with either transformed basic authentication credentials (when `USE_WIREPACT=false`) or with the presented access token (`USE_WIREPACT=true`). Either way, the legacy API receives basic authentication credentials in the form of `<username>:<password>` and returns the data that is then presented to the user.

To install and run the case study without any interference of the operator or the rest of the solution, follow the installation guide in the readme on https://github.com/WirePact/poc-showcase-app. To install and use the whole POC, please refer to the installation guide in the Appendix.

4.7.2 Automation Engine for Applications

As explained in the abstract section about the architecture, the automation engine is generally optional. If omitted, the user is responsible for configuring the proxy and the translator. In the POC, the automation engine is a Kubernetes operator written with the .NET SDK in C#. The source of the POC operator is hosted on GitHub: https://github.com/WirePact/poc-operator. The operator (automated and customized management of resources in Kubernetes, see Section 2.3) intercepts events for `Deployments` and `Services`. To update services and deployments in the POC, an annotation (basically a key-value storage in the metadata of an object in Kubernetes) is used. In future work, the operator may react to Custom Resource Definitions (CRD) as well.

Figure 4.14 gives an overview of the process that an event of the Kubernetes API completes. When the operator is notified by Kubernetes that a service or a deployment was created or modified, the operator determines the type and uses the specific controller to reconcile

[6] https://docs.microsoft.com/en-us/aspnet/core/razor-pages/

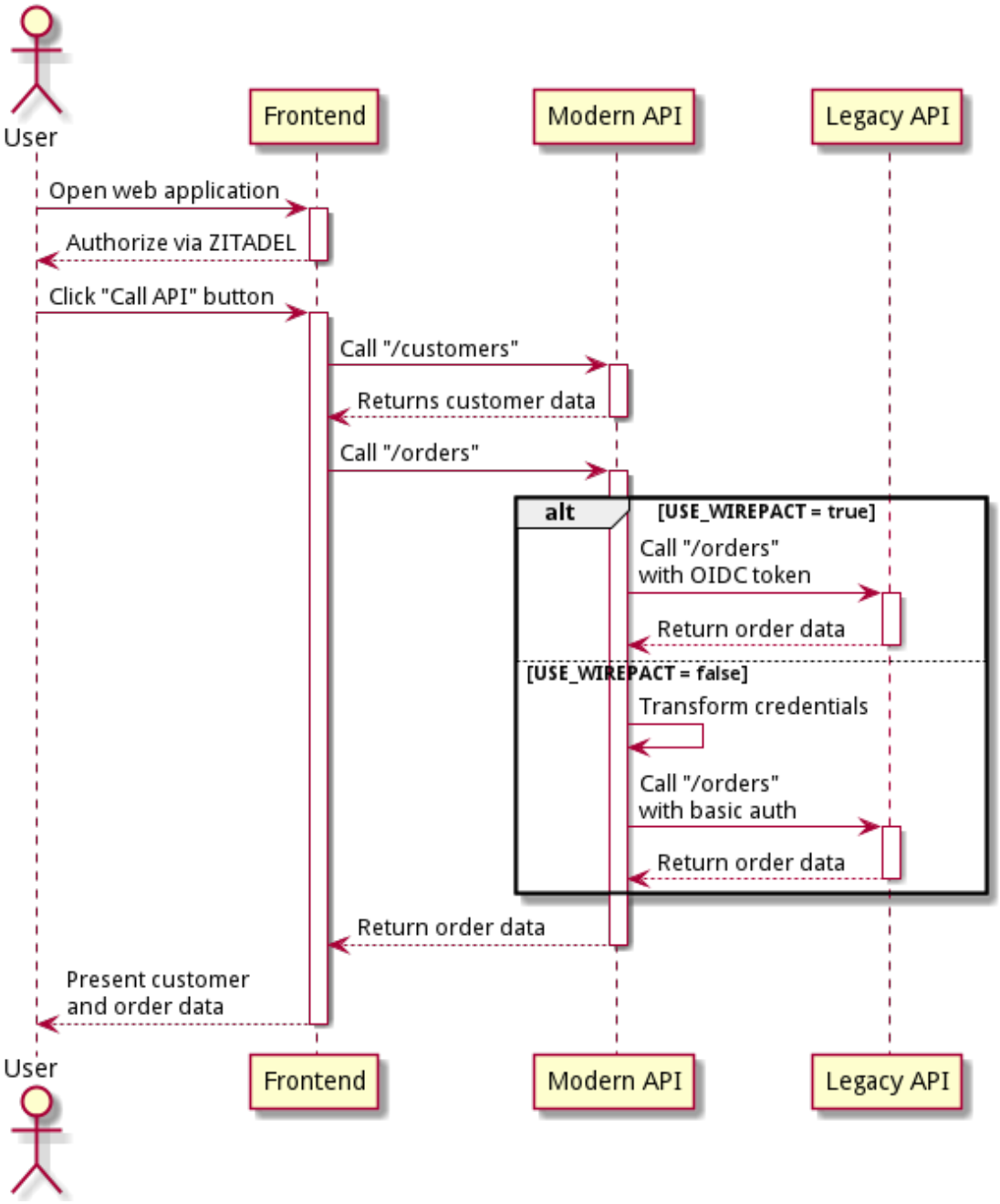

Figure 4.13: Sequence Diagram of the Communication in the Case Study

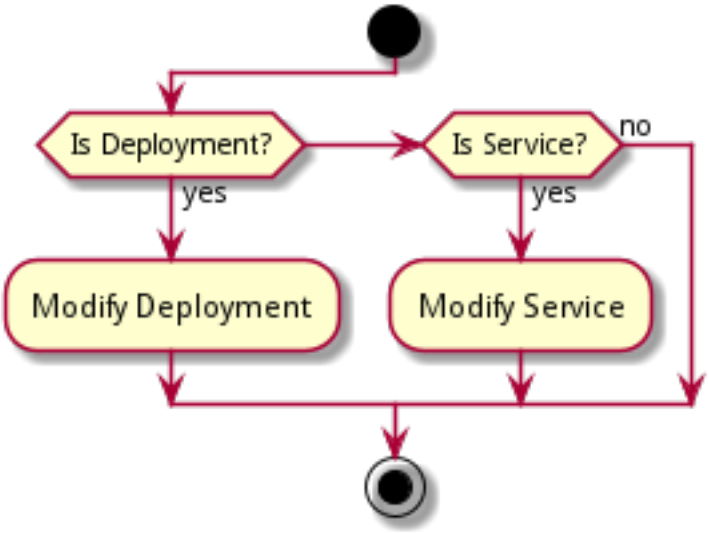

Figure 4.14: Activity Model for Kubernetes Resources in the Automation Engine

the resource. If the entity is a deployment and it is relevant for the authentication mesh, the operator will modify the deployment. On the other hand, if the entity is a service, the operator modifies the service if it is part of the mesh.

In the case of a deployment, Figure 4.15 shows the process for the management-event of the deployment. The first step of the operator is to determine if the entity is relevant for the authentication mesh. If the deployment contains the annotation `ch.wirepact/port` in its metadata, it is automatically part of the mesh. If the deployment is already configured, further reconfiguration is skipped. If not, the operator fetches the already configured ports of the deployment, and generates two additional ports. One port is used for the Envoy sidecar while the other is configured for the translator sidecar. The next step is to generate and store the Envoy configuration in a Kubernetes `ConfigMap`. Last, the sidecars are injected into the deployment configuration and the Kubernetes client stores the modified manifest.

When reconciling a service, Figure 4.16 shows the activities of the operator during the reconciliation. The service counts as relevant if the annotation `ch.wirepact/deployment` is attached in the metadata of the service. The value of this annotation gives the deployment object to which the service should point. Then, the operator reads the annotations on the service to determine the port in question and searches for the port in its manifest. Then the port will receive a new “target port” that points to the Envoy port of the deployment. Last, the Kubernetes client will store the changed service.

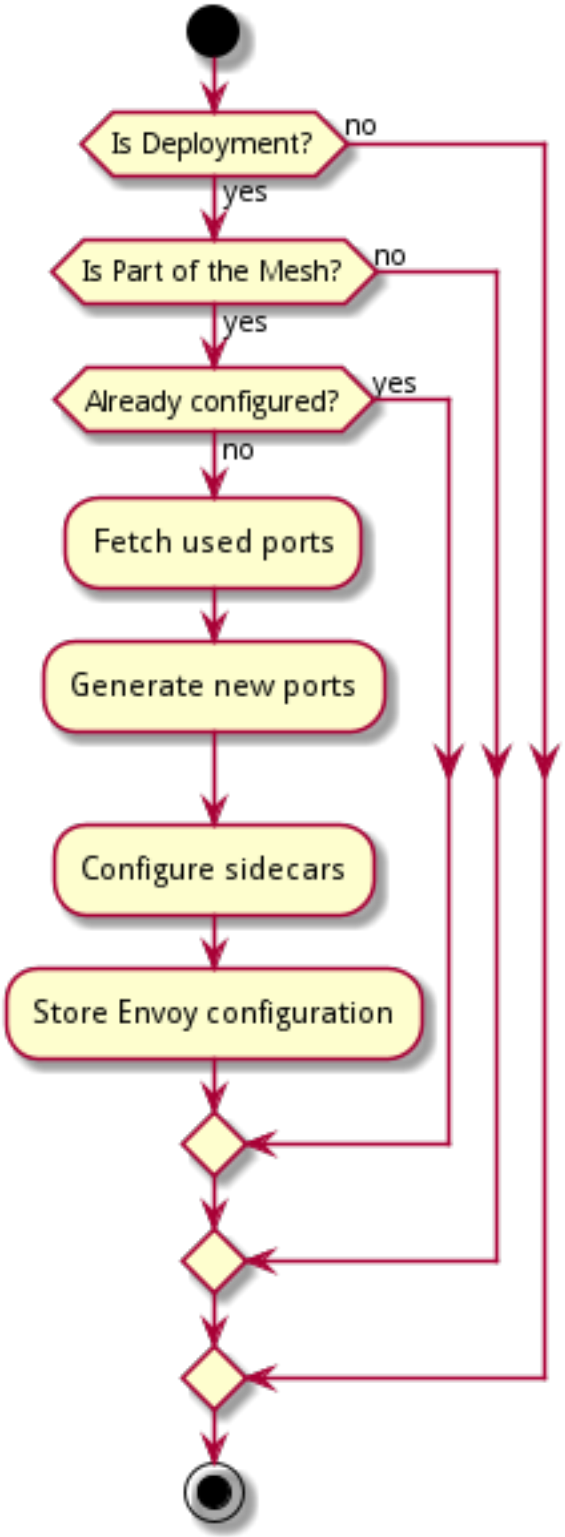

Figure 4.15: Automated Configuration of a Kubernetes Deployment in the POC

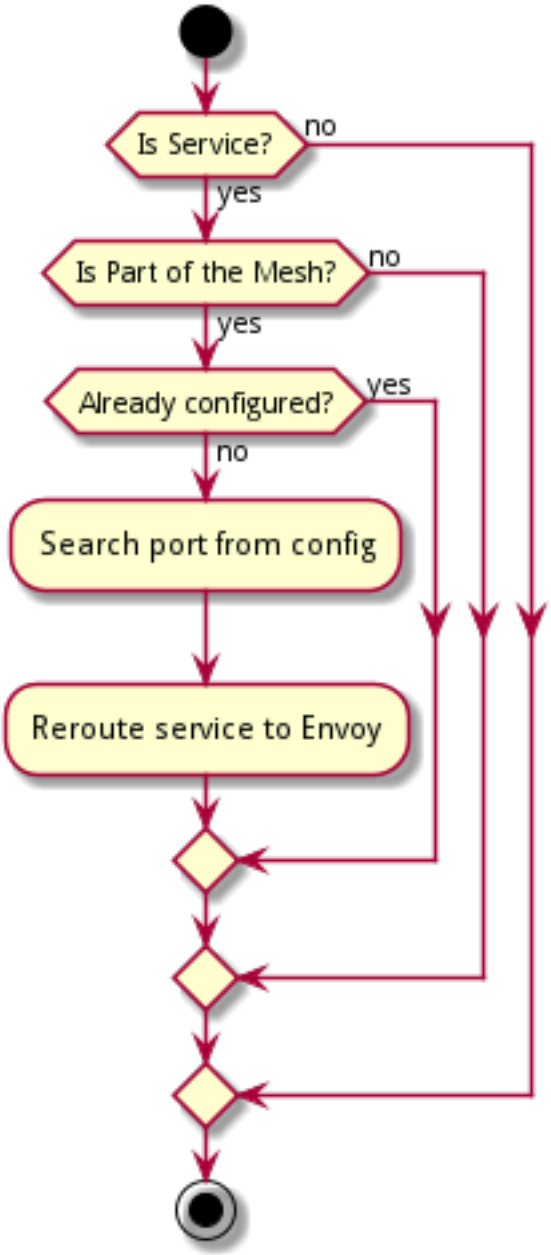

Figure 4.16: Automated Configuration of a Kubernetes Service in the POC

4.7.3 Network and Routing Proxy for Communication

In the POC, the proxy sidecar is an Envoy proxy with its configuration injected by the automation engine. The operator injects the sidecar whenever a `Deployment` is created or updated via the Kubernetes API. The operator attaches the proxy and adds several annotations that are used for communication with a `Mutation Webhook`. Furthermore, a `ConfigMap` with the envoy configuration is created during the webhook.

Two parts of the envoy configuration are crucial. First, the `filter_chain` of the inbound traffic listener contains a list of `http_filters`. Within this list of filters, the external authorization filter is added to force Envoy to check if an arbitrary request is allowed or not:

```
# ... more config
http_filters:
  - name: envoy.filters.http.ext_authz
    typed_config:
      '@type': type.googleapis.com/
        envoy.extensions.filters.http.
        ext_authz.v3.ExtAuthz
      transport_api_version: v3
      grpc_service:
        envoy_grpc:
          cluster_name: auth_translator
        timeout: 1s
      include_peer_certificate: true
  - name: envoy.filters.http.router
# ... more config
```

Second, the external authorization service must be added to the `clusters` list to be access via the configured name (`auth_translator`):

```
# ... more config
- name: auth_translator
  connect_timeout: 0.25s
  type: STATIC
  typed_extension_protocol_options:
    envoy.extensions.upstreams.http.v3.HttpProtocolOptions:
      '@type': type.googleapis.com/
        envoy.extensions.upstreams.http.
        v3.HttpProtocolOptions
      explicit_http_config:
        http2_protocol_options: {}
  load_assignment:
    cluster_name: auth_translator
```

```
    endpoints:
      - lb_endpoints:
          - endpoint:
              address:
                socket_address:
                  address: 127.0.0.1
                  port_value: <<PORT_VALUE>>
# ... more config
```

This configures Envoy to find the external authorization service on the local loopback IP on the configured port. Since the transformer uses gRPC (`grpc_service: envoy_grpc: ...` in the filter config), http2 must be enabled for the communication. In a productive environment, timeouts should be set accordingly.

4.7.4 Translator

The translator is the part of the POC that shows the modification of HTTP headers per request. Since the intermediate DSL is not implemented in the POC, the translator converts an access token to static basic authentication credentials. If any error occurs or the translator call exceeds ten seconds, Envoy returns a HTTP 403 Forbidden message by default. The source code resides on GitHub: https://github.com/WirePact/poc-demo-translator.

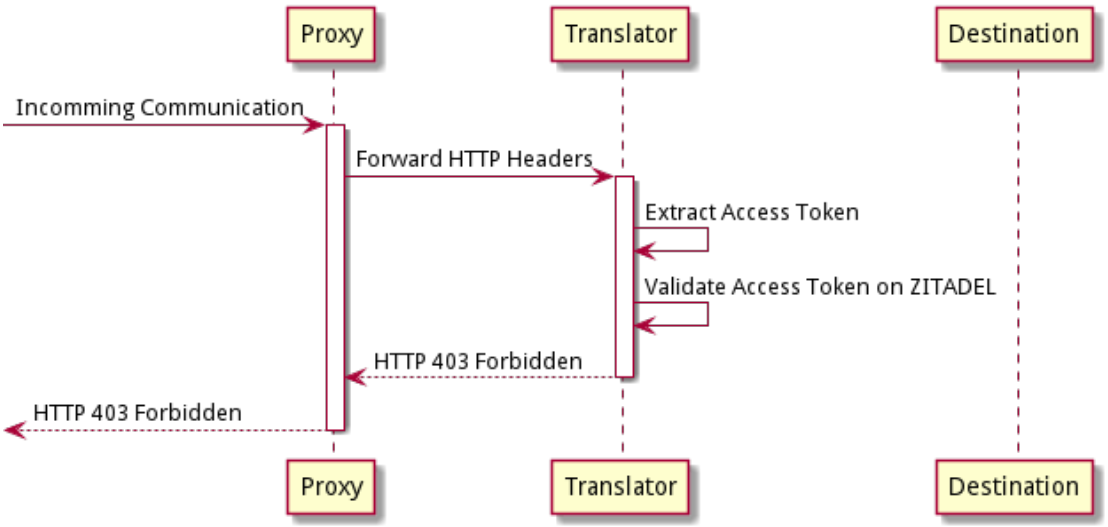

Figure 4.17: Communication with an invalid access token

Figure 4.17 shows the sequence for an access token that is not valid. Envoy forwards the HTTP headers to the translator which extracts the `Authorization` header. If it is not a `Bearer` access token, or if the validation with ZITADEL fails (if the token is not

valid or expired), the translator returns an `Unauthorized` (HTTP 401) or `Forbidden` (HTTP 403) response depending on the status. The `Unauthorized` status is returned when no access token is provided (i.e. the HTTP header is missing) and `Forbidden` is the response, if the token is invalid. In either case, Envoy will return the returned status code to the caller and the call ends. The destination application does not receive any communication or notification about this event.

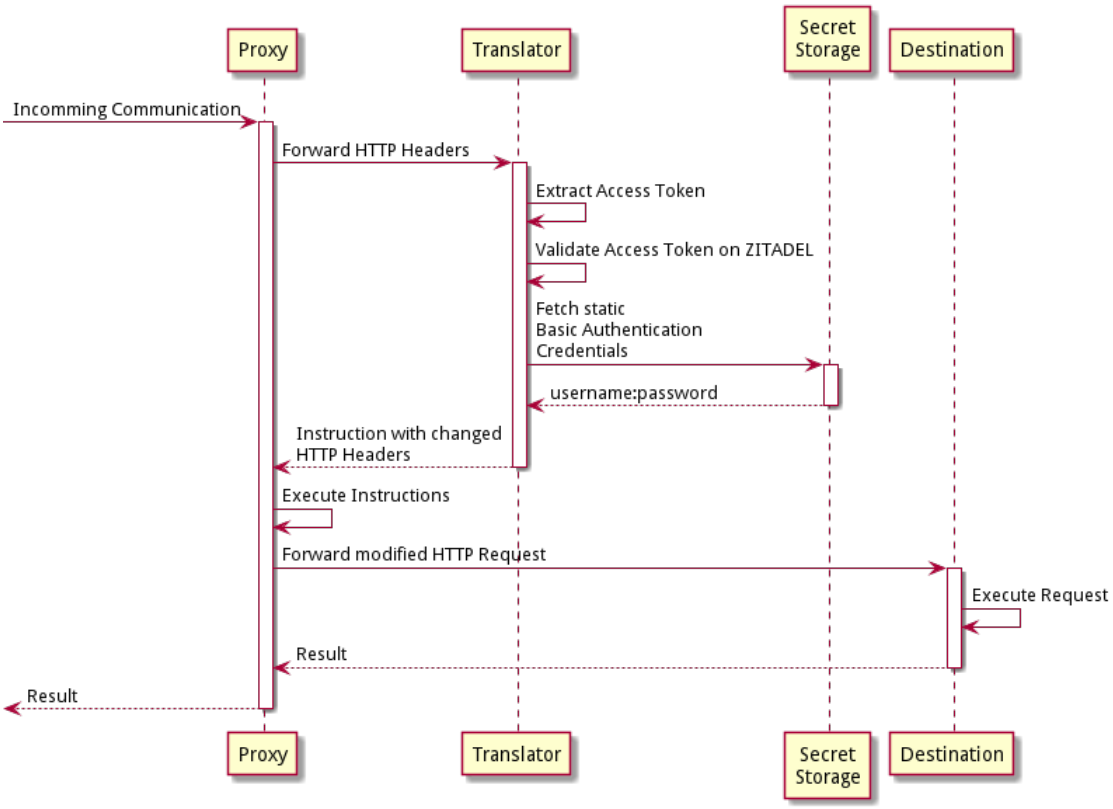

Figure 4.18: Communication with a valid access token

In contrast to Figure 4.17, the sequence in Figure 4.18 shows the success path of a communication. If the given access token is valid, the translator fetches the static Basic Authentication credentials (i.e. username and password) from the secret storage. The secret storage in the POC is a simple Kubernetes Secret. The received credentials are then transformed in the correct encoded Basic Authentication format (as described in RFC7617). Afterward, the translator returns an instruction-set for Envoy to process the HTTP request. Envoy executes the instruction and forwards the call to the destination and returns the response - if any.

4.7.4.1 Instructions for Rejected Request

When the translator decides that the request is unauthorized or forbidden, it returns a `DeniedResponse` to Envoy. The reponse is encoded in a binary "Protocol Buffers"[7] format, but a JSON example would be:

```
{
  "deniedResponse": {
    "status": {
      "code": 403
    },
    "body": "No valid Authorization provided"
  }
}
```

There are additional fields encoded in the message, but the example above shows the essential parts of the denied response.

4.7.4.2 Instructions for Accepted Request

In contrast to the rejected response instructions, an accepting response may include modifications for HTTP headers. It is possible to add new, modify, and remove headers from the request for the upstream (i.e. the destination of the request), as well as adding additional or modifying headers for the downstream (i.e. the source of the request when the result is returned). Such a response that replaces the `Authorization` header with the basic credentials is:

```
{
  "okResponse": {
    "headers": [
      {
        "header": {
          "key": "Authorization",
          "value": "Basic Q2hyaXN0b3BoQnVlaGxlcjpTdXBlclNlY3VyZQ=="
        }
      }
    ]
  }
}
```

In the response above, if an `Authorization` header already exists, it is replaced. Otherwise, the value is added. In the case of the distributed authentication mesh, this

[7] Binary Data Format by Google: https://developers.google.com/protocol-buffers

technique can be used to consume the user identity (i.e. remove a custom header) and add the specific authentication credentials for the upstream.

5 Evaluation

TODO: show in an evaluation, that the provided solution is working and improves the situation

6 Conclusion

TODO : Conclusion

TODO : Further work

Bibliography

Bryan, Paul, and Mark Nottingham. 2013. “Javascript Object Notation (JSON) Patch.” RFC. Internet Engineering Task Force IETF. https://tools.ietf.org/html/rfc6902.

Burns, Brendan, and David Oppenheimer. 2016. “Design Patterns for Container-Based Distributed Systems.” In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns.

CNCF. 2021. “Kubernetes Website.” *GitHub Repository*. https://github.com/kubernetes/website; GitHub.

Cooper, Dave, Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, and Stephen Farrell. 2008. “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.” 5280. Internet Engineering Task Force IETF. https://doi.org/10.17487/RFC5280.

Creative Commons. 2021. “Attribution 4.0 International (CC BY 4.0).” https://creativecommons.org/licenses/by/4.0/.

Curbera, F., M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. 2002. “Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI.” *IEEE Internet Computing* 6 (2): 86–93. https://doi.org/10.1109/4236.991449.

Dobies, Jason, and Joshua Wood. 2020. *Kubernetes Operators: Automating the Container Orchestration Platform*. " O’Reilly Media, Inc.".

F5 Inc. Authors. 2021. “Authentication Based on Subrequest Result.” *NGINX*. https://docs.nginx.com/nginx/admin-guide/security-controls/configuring-subrequest-authentication/.

Hardt, Dick, and others. 2012. “The OAuth 2.0 Authorization Framework.” RFC. Internet Engineering Task Force IETF. https://tools.ietf.org/html/rfc6749.

Istio Authors. 2021a. “External Authorization.” *Istio*. https://istio.io/latest/docs/tasks/security/authorization/authz-custom/.

———. 2021b. “Mutual TLS Migration.” *Istio*. https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/.

Jones, Michael B., Bradley John, and Nat Sakimura. 2015. “JSON Web Token (JWT).” RFC. Internet Engineering Task Force IETF. https://tools.ietf.org/html/rfc7519.

Kratzke, N., and R. Peinl. 2016. “ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects.” In *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, 1–10. https://doi.org/10.1109/EDOCW.2016.7584353.

Li, W., Y. Lemieux, J. Gao, Z. Zhao, and Y. Han. 2019. “Service Mesh: Challenges, State of the Art, and Future Research Opportunities.” In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 122–25. https://doi.org/10.1109/SOSE.2019.00026.

Montesi, Fabrizio, and Janine Weber. 2016. “Circuit Breakers, Discovery, and API Gateways in Microservices.” *CoRR* abs/1609.05830. http://arxiv.org/abs/1609.05830.

Naik, N., and P. Jenkins. 2017. “Securing Digital Identities in the Cloud by Selecting an Apposite Federated Identity Management from SAML, OAuth and OpenID Connect.” In *2017 11th International Conference on Research Challenges in Information Science (RCIS)*, 163–74. https://doi.org/10.1109/RCIS.2017.7956534.

Nielsen, Henrik, Roy T. Fielding, and Tim Berners-Lee. 1996. “Hypertext Transfer Protocol – HTTP/1.0.” 1945. Request for Comments. RFC 1945; RFC Editor. https://doi.org/10.17487/RFC1945.

Reschke, Julian. 2015. “The ’Basic’ HTTP Authentication Scheme.” RFC. Internet Engineering Task Force IETF. https://tools.ietf.org/html/rfc7617.

Rose, Scott, Oliver Borchert, Stu Mitchell, and Sean Connelly. 2019. “Zero Trust Architecture.” National Institute of Standards; Technology.

Sakimura, Natsuhiko, John Bradley, Mike Jones, Breno De Medeiros, and Chuck Mortimore. 2014. “Openid Connect Core 1.0.” Spec. The OpenID Foundation OIDF. https://openid.net/specs/openid-connect-core-1_0.html.

Appendix A: Teaching Material for Kubernetes Operators

TODO