UiO : Department of Informatics
University of Oslo

Authentication in the mesh with WebAssembly

Sondre Halvorsen

Master's Thesis, Spring 2021

Thesis submitted for the degree of Master in Informatics: programming and system architecture

30 credits

Department of Informatics Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Master Thesis

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

At the start of the last decade Marc Andreessen stated in his now famous blog entry; 'Software is eating the world' [173], and as software is eating the world, problems stemming from its security, or lack thereof, is eating it as well. Organisations are increasingly moving to the cloud and adopting new architecture patterns in this new environment, such as cloud native and microservice architecture. While moving to the cloud generally results in better security for organisations, due to large professional platforms, microservice architectures introduce some new problems in regards to cross-cutting concerns like security, robustness and observabiltity. Service mesh is a technology that aims to provide cloud native and application agnostic solution to many of the challenges with micro service architectures. In parallel with the cloud native revolution there has been innovations in areas like security as well. Authentication, authorization, and access control have changed drastically, with new requirements for how users want to manage and use their identity. *Zero Trust Architectures* is an example of this drawn to its logical conclusion where no connection is trusted by default. Unfortunately security breaches stemming from poor implementation of security protocols and frameworks rank among the highest still. As the trend is to divide a system into more and more fine grained services, the issue with poor implementation of authentication, authorization and access control are likely to rise. This thesis, in collaboration with Norwegian Labour and Welfare Administration (NAV), explores a approach to utilize service mesh technology to improve the authentication and authorization problem in a microservice architecture, and make it easier to implement Zero Trust Architectures. The approach builds upon recent innovations in the service mesh space with support for extending data plane proxies with WebAssembly extensions. This thesis explores different approaches for an organisation to utilize service meshes for authentication and authorization, implements a working prototype for extending data plane proxies with authentication capabilities using WebAssembly, and compares the prototype against the alternatives performance, security and operational characteristics.

Acknowledgements

The thesis author wishes to thank Nils Gruschka for his support as a thesis supervisor. Thank you for the discussions and feedback during the thesis work. Also a big thanks to the external supervisor Audun Fauchald Strand, and the NAIS [106] team at Norwegian Labour and Welfare Administration (NAV), for the providing a real world problem to base the thesis on. Audun and the NAIS team has during the thesis supported the work with feedback, source code review and technical discussions. The support from NAV and the NAIS team has been above and beyond the support expected of a external thesis collaborator.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Every Business is a Software Business

Watts S. Humphrey Winning with Software: An Executive Strategy (2001)

1.1 Motivation

The world is becoming increasingly driven by software interconnected over the internet, and organisations and teams are working hard to meet the increasing expectations of customers and users for digital services. There are several important shifts in the computing landscape that are driven by this increased pressure, but from a sociotechnical perspective arguably the largest shift is from traditional enterprise software development to *Cloud native* [92] development. Cloud native is an ill-defined term in the academic sense, but the core concept is changing the way organisations construct critical business systems by using new platforms and development practices that utilize new possibilities driven by cloud technology (see 2.3). For an example of the benefits look no further than to The Norwegian Labour and Welfare Administration (NAV), which made the decision to move to a cloud native way of developing software. The software in question is delivering critical services for thousands of Norwegians every month. As the graph in figure 1.1 shows the average number of application deployments increased dramatically, by 1290 percent from 2014 to 2021. Research has shown that deployment frequency and batch size is correlated with high performing organisations [35], meaning speedy development and frequent changes does not need to impact quality or reliability, in fact for high performance organisations it improves both. The results for NAV has been awards and increased customer satisfaction with NAV's services [115]. Results like these are what motivates organisations across the world to adopt the cloud and cloud native development.

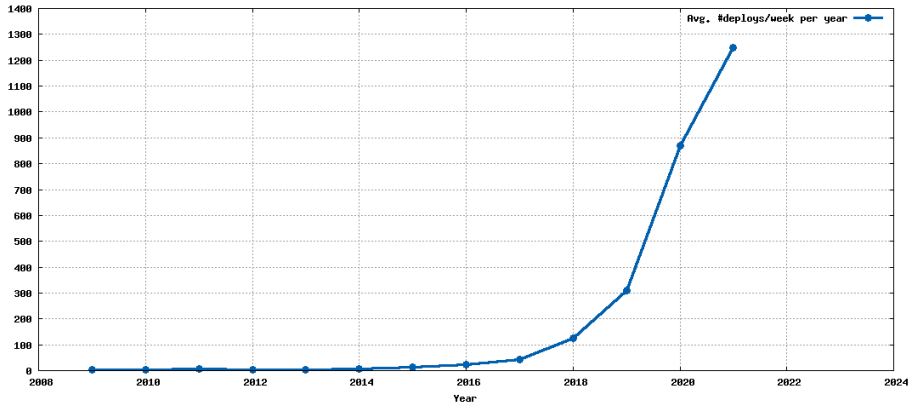

Figure 1.1: Average number of application deployments per week by year at NAV (Source: [108])

New Technology For Connecting Services Cloud native architecture lends itself to service-oriented architecture where components of a larger system communicate over a network. An increasingly common architectural paradigm within the cloud native space is *microservices* [77], see section 2.5. Microservices have many beneficial properties like technical and organisational scalability. However there are drawbacks organisations associated with utilizing microservice architecture. Splitting a system over an asynchronous network makes the system a distributed system, with all the errors and failure modes a distributed system entails. Traditional systems, built as monolithic applications running as a single process on a single machine could leverage operating system and hardware supplied guarantees for important aspects, such as consistency, observability and security. Luckily the cloud native community have developed new technical solutions to mitigate the issues, and make it easier for organisations to develop many small services that connect over the network, even maintaining a ratio of 10:1 of services to developers [68]. This category of technology is aptly named *service mesh*. Service meshes aim to provide observability, reliability and security to microservice architectures (see 2.6). What is important is that cloud native organisations are moving to adopt service meshes into their infrastructure [1], and the adoption is creating value for the organisations [25]. This has resulted in a lot innovation in the service mesh space. However an issue that has proved resilient is how to extend service mesh implementations with custom behaviour. Organisations often have unique operational environments that require custom integration across platforms, and third-party providers are eager to enter the service mesh market with plugins and extensions. Different service mesh implementations have attempted different approaches to letting users extend their service mesh solutions, like Istio's *Mixer* [57]. The main issue with the attempts have been that the extensions are costly in terms of performance and with reliability problems The extensions problem has remained a

thorn in the service mesh landscape until recently. WebAssembly, a secure, fast and sandboxed virtual machine born from the attempt to introduce compiled languages into the browser has been introduced as a promising way to extend service meshes (see 2.8). Instead of extending by adding an intercepting service to the mesh proxies, WebAssembly modules can be ran directly in the proxies serving the traffic in the service mesh. Several organisations are signaling their support like Google and Solo.io [58]. In October 2020 the Envoy proxy became the first service mesh proxy to support WebAssembly extensions [56] by default. There is also work being done on standardizing an Application Binary Interface (ABI) so proxy extensions modules can be reused across service meshes providing a standard way of developing extensions that can be reused across service mesh implementations (see 2.7.5). An important questions that needs to be answered however is on maturity. WebAssembly extensions approach is new and with questionable maturity, there is little research into how such extensions can best be leveraged by organisations.

Zero Trust When operating on-premise digital infrastructure it is common to utilize perimeter network models [6], however, when an organisation moves to the cloud, and especially when the organisation is in the migration phase, the perimeter network model becomes hard to maintain, if not insecure. Organisations moving to the cloud are increasingly implementing *Zero Trust Network architecture* [6]. Zero Trust Networks can be implemented in different ways and in different levels of the network stack. While network level *Zero Trust Architectures* (see 2.11) are to most common, token-based Zero Trust Architecture is an application level Zero Trust Architecture. The approach uses security tokens like JSON Web Tokens (JWT), OAuth 2.0 *On-Behalf-Of* flows [93] to create dynamic, and secure application level commutation between applications and service. While Zero Trust Architecture is establishing itself as a the way to secure enterprise communication [153], how best to implement is still developing.

Old Vulnerabilities Still Cause Problems At the same time organisations all over the world are moving to the cloud [3], secure authentication and authorization, have improved with protocols such as OAuth 2.0. and OpenID Connect [32]. These protocols are better suited for today's connected world [85]. Complexity stemming from new protocols is being tackled by organisations and open source communities by creating tooling and software libraries that make it easier to develop correctly with the new protocols [97]. Regardless of the new innovations, flawed implementation of authentication and authorization remain amongst the most common software security issues today [174].

New Possibilities There are many ongoing efforts looking into the security possibilities that service meshes open up. Projects such as SPIFFE [148] allow service mesh implementations to bootstrap identity for cross-workload authentication and many data plane proxies support integrating custom authorization services [28]. Pushing the responsibility for executing security activities into

the service mesh could be a great boon for security and means it would be possible to implement the code for such processes once, and still be able to distribute the execution across the service mesh, avoiding single-point of failure problems inherent in traditional authentication and authorization services. Service mesh technology together with the recent advances in extensibility, by way of WebAssembly, is an opportunity for a new cloud native approach to Zero Trust Architecture. Building on-top of the already available cloud native infrastructure, a cloud native Zero Trust Architecture approach could enable more organisations to securely use the cloud and for users to experience more secure services. This theis will investigate this space, and through implementing actual components, try to expand the insight in this area. Finally does moving security processes into the service mesh reintroduce problems with development teams not caring about security? A big part of the agile, DevOps and cloud native movements has been the idea that developers should be exposed to how their products run, making the feedback loop short, resulting in better, more robust and more secure products. If we are to use the service mesh for security Zero Trust Architecture the new solutions should not repeat old mistakes and aim to keep developers in the loop about the security of the products they are developing.

1.2 Contribution

The main goal for this thesis is to explore how token-based Zero Trust Architecture can be implemented as part of a the service mesh for development teams. This thesis focuses on authentication and delegated authorization components of Zero Trust Architecture. The Service mesh represents a new and promising platform to create application level zero trust architectures and has the potential for providing organisations control over security, while still keeping developers in the loop regarding security of their upstream applications. The thesis contribution, which separates it from other works in this space is to look at implementing the authentication in the data plane of the service mesh. The benefits and drawbacks of this approach will be discussed in sections 3, 4 and 6.

Proof of Concept Filter This thesis will systematically analyse different methods of incorporating authentication and authorization into the data layer of the service mesh. Further, the approach that is best suited for the requirements from the thesis collaborator (Norwegian Labour and Welfare Administration) will be selected and implemented as a proof of concept, see section 3 for more on the requirements that will guide the selection of approach. Lastly the implementation will be evaluated in regards to performance, and the security benefits and trade-offs will be analysed, see section 6 for more on testing, evaluation and the analysis of the proof of concept implementation.

1.3 Research Questions

The thesis aim to explore how organisations can build token-based Zero Trust Architecture using service mesh capabilities. Additionally just because implementing such architectures is possible, can it be done in line with cloud native, agile development and organisational practices.

- RQ1: What approaches exists for handling authentication and/or authorization in common service mesh implementations?
- RQ2: How can this be implemented for performance in regards to latency, security capabilities and operational complexity for the serviced application?
- RQ3: How can this be implemented in a way that improves the security, while still keeping developers in control of the supported applications security, in line with cloud native development practices?

Chapter 2

Background

2.1 Definitions

Some concepts and terms will be used throughout the thesis, and to remove ambiguity the way they are used in the thesis is defined here.

Application	Program or group of programs designed for end users.
Service	Program, or group of programs, primarily designed to be used by other services or applications.
Risk	A potential problem calculated as: impact times likelihood of occurring.
Threat	A path in the system to a risk occurring.
Vulnerability	A weakness which can be exploited by a threat.
Mitigation	A countermeasure against a threat, something that can be done to prevent or at least reduce the chance of threat to succeed.
Runtime engine	Program that convert original source code, or intermediate languages, into machine code which it executes. In addition runtime engines provide common routines and functions that applications targeting the engine can utilize.
Authentication	A process that, when successful, confirms the identity of an entity (e.g., user, organisation, application, service).

Authorization	A process that verifies whether an entity can access a resource or perform some action.
AuthN	Common abbreviation of Authentication.
AuthZ	Common abbreviation of Authorization.
CI/CD	In software engineering, CI/CD is the combined practice of continuous integration and either continuous delivery or continuous deployment.
CI	Continuous Integration, the practice where a development team continuously integrates small changes to the software, in contrast to large intermittent integration work. It usually involves using testing frameworks and automatic build and test platforms composing a CI pipeline.
CD	Continuous Delivery or Continuous Deployment, the practice of keeping the code and artifacts built from the code in a ready-to-deploy state at all times or in the case of Continuous Delivery, deploying software to production on all changes to the software that passes through the CI pipeline.
Upstream	In proxy terminology an upstream host receives connections and requests from a proxy and returns responses.
Downstream	In proxy terminology a downstream host connects to a proxy, sends requests, and receives responses from the proxy or upstream services.
Ingress	Requests crossing a network boundary into a private network from a untrusted or less trusted network zone.
Egress	Request originating from inside a private network that crosses the network boundary into untrusted network zones.

Table 2.1: Concepts and Definitions

2.2 Cloud Computing

Cloud computing has become an ubiquitous term, cloud computing here refers to what The National Institute of Standards and Technology (NIST) describes as:

> *cloud computing enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.* (Source: [88])

Simply put cloud computing enables an organisation to treat computational power, storage, network and common application services as an utility where usage can be tuned dynamically and in real time. Use of cloud computing is increasing, also for government organisation [3]. Like private businesses, government organisations are also being demanded more when it comes to digital services [19]. Cloud computing is seen as an enabling step, allowing government organisations to focus on developing services and applications for citizens.

The cloud computing market is categorized after the division of responsibility between the customer and provider in the product that is offered.

- **Software-as-a-service (SaaS)** involves delivering software as a service offering to customers. Licenses are typically provided through a pay-as-you-go model or on-demand. This type of system can be found in Microsoft Office's 365 or Dropbox.
- **Infrastructure-as-a-service (IaaS)** involves a method for delivering everything from operating systems to servers and storage and networks as part of an on-demand service. Customers do not need to purchase and install servers, and instead they can procure compute resources in an outsourced, on-demand service. Popular examples of *IaaS* are Google Cloud, AWS and Azure.
- **Platform-as-a-service (PaaS)** is considered the most complex of the three layers of cloud-based computing. *PaaS* shares some similarities with *SaaS*, the primary difference being that instead of delivering software online, it is actually a platform for creating software that is delivered via the Internet. This model includes platforms like Salesforce.com and Heroku.

2.3 Cloud Native

Organisations that were created after cloud computing became a thing as often called *born-in-the-cloud companies*, which means the products and services they provide were built from scratch on cloud platforms. Cloud platforms enable organisations to create applications and services that are easier to change according to customer demand, that are more observable, easier to fix and more

scalable. Applications and services that are created to take advantage of the cloud are called *cloud native*. The term is not exclusive to public cloud based deployment, the core ideas and principles behind cloud native development can, and have, been used on-premise as well.

The Cloud Native Computing Foundation defines the term cloud native like the following:

> *Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.* (Source: [13])

2.3.1 The Twelve-Factor App

Several communities in the early days of cloud computing realised that applications and services should be developed differently in a cloud environment. For instance, on-premise scaling is often done vertically, by adding more resources like memory or upgrading the CPU, in the cloud on the other hand, scaling is done horizontally, by adding more machines and replicating the application across the machines. Scaling horizontally favors *stateless applications*, applications that do not hold any permanent state on its own. The Twelve-Factor App principles were first developed by engineers at Heroku[45], a turn-key cloud platform, and their The Twelve-Factor App manifesto[175] describes twelve best practice principles for developers wanting to develop applications to be ran in the cloud.

1. **Codebase: One codebase tracked in revision control; many deploys.** An Application has one codebase, but can be built and deployed into many different environments like dev, test and prod.

2. **Dependencies: Explicitly declare and isolate dependencies.** Applications should use dependency managers that explicitly declare and version dependencies, container technologies like docker makes it possible to declare operating system dependencies. This reduces difference between deployment environments.

3. **Configuration: Store configuration in the environment.** Configuration data should be separated out from application code, and configuration should be injectable through configuration files, environment variables etc.

4. **Backing Services: Treat backing services ad attached resources.** External services such as caches and databases should be accessed trough configuration.

5. **Build, Release, Run: Strictly separate build and run stages.** Aim to fully automate build and release stages through CI/CD practices.

6. **Processes: Execute the application in one or more stateless processes.** Data should be saved outside of the application process. This enables compute elasticity.

7. **Data Isolation: Each service manages its own data.** Also a key tenet in microservices, see section 2.5. This means other services cannot directly access the data of another service.

8. **Concurrency: Scale out via the process model.** Scaling of cloud native applications is preferably done horizontally, this improves scalability and resource utilization.

9. **Disposability: Maximize robustness with fast startup and graceful shutdown.** Applications should be fast to start up and should be safe to stop using operating system signals such as SIGTERM [128].

10. **Dev/Prod Parity: Keep development, staging, and production as similar as possible.** Hard to reproduce bugs are easy to introduce into a working systems if the application under test in a dev environment differs from the application running in production. Containers make it easier to package applications into artifacts that can be ran identically in every environment.

11. **Logs: Treat logs as event streams.** Cloud native systems are distributed systems with all the problems a distributed system entails [134]. Sufficient observably into the system is necessary to manage and develop such systems.

12. **Admin Processes: Run admin and management tasks as one-off processes.** Administrative tasks should be executed as short lived processes. An example of this could be a database migration, such migrations should be run in the production environment and the code for the migration should be shipped with the application release.

2.3.2 Containers

Containers are a lightweight virtualization approach where guest systems run on the host system's kernel. While the container technologies that are used widely today are relative recent technologies, Docker (2013) [21], LXC (2008) [48], the technology and ideas that underpin them are much older. The idea behind containers is to let applications and services run on an operating system without interfering with other processes, effectively isolating processes from each other. The Linux operating system facilitates this through two key kernel features:

- **Namespaces** [65]. Enable processes to have different views of the system than other processes, without resorting to more heavyweight constructs such as virtual machines. Things that can be namespaced include file system, network and user space. The idea of isolating what processes

can "see" is not new. FreeBSD introduced *Jails*[64] and Solaris created *Zones*[126] to solve similar problems, such as the all-mighty power of root, that containers would later tackle as well.

- **Control groups (cgroups)** [89]. Control groups allow for fine-grained control over system resources, such as CPU time, memory, and IO. It would be hard to create the illusion of running alone if a container could hog all the system resources, starving the neighboring containers.

The Container Abstraction Namespace and control groups are kernel features and are not designed to be interacted with in a user friendly fashion. Container technologies like Docker create an abstraction over namespace and control group handling, exposing a more user friendly interface for developers to create applications in. Docker mainstreamed containers by wrapping the container abstraction together with tooling for building and distributing *container images* [116]. For cloud native development containers have become the "unit of deployment", and unlike other popular deployment formats like JAR files[119] container can run any program that can run on Linux.

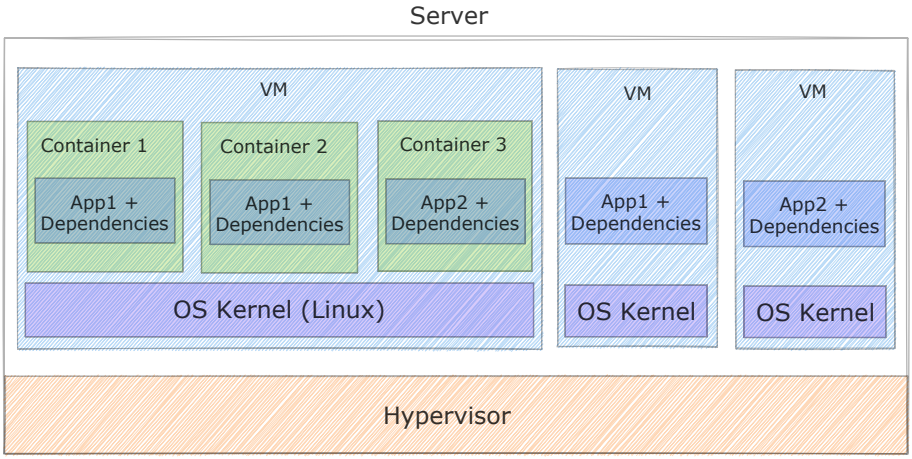

Figure 2.1: Compression of containers and virtual machines

2.3.3 Orchestration

Containers are a great technology for creating a "unit-of-deployment" in cloud native development. But when the scale increases from a handful of containers running together, to hundreds of containers, orchestration is necessary. At the time of writing Kubernetes [72] is by far the most popular orchestration technology [34]. Kubernetes is an open source container orchestration platform that enables applications to be released, scaled up or down, and updated without downtime. Kubernetes pluggable and customizable platform provides a basic

platform with a rich set of built-in features, but it allows developers building on top of it to extend the functionality through third-party or self built extensions.

Kubernetes provides the following features:

- **Service discovery and load balancing** DNS and IP based routing in the cluster and can distribute network load over a deployment.
- **Storage orchestration** Mounting of storage systems like local storage, public cloud provided storage, buckets etc.
- **Automated rollouts and rollbacks** through a declarative configuration Kubernetes changes the state of the cluster to mach desired state, for example by creating new containers, removing containers, mounting storage systems or changing access policies.
- **Automatic bin packing** Running on a cluster of Nodes Kubernetes can distribute running containers over the available nodes for the best fit.
- **Self-healing** Handles failing containers through user defined health checks.
- **Secret and configuration management** Lets developers store and manage secrets and sensitive information that can be injected into containers without the need to rebuild container images or adding secrets to the images themselves.

Kubernetes Cluster

The top-level component in Kubernetes is the cluster. When you deploy Kubernetes you get a cluster. A Kubernetes cluster runs on a set of worker machines called nodes, each cluster has at least one node. A Kubernetes cluster is divided into two main components, the control and worker nodes, which both have several sub-components. The control plane components make global decisions about the cluster, the worker nodes host the containers with applications and services. Figure 2.2 shows the different sub-components that make up a worker node. The *kublet* process is responsible for making the control planes decisions a reality on the node. The *kube-proxy* process is responsible for networking between pods in the cluster and connections to outside the cluster. The container runtime, usually *containerd* is responsible for starting, stopping and managing the containers.

Kubernetes Core Concepts

Inside a Kubernetes cluster there are some core concepts and abstractions that govern the state of the cluster. Figure 2.3 shows a diagram of how these abstractions connect.

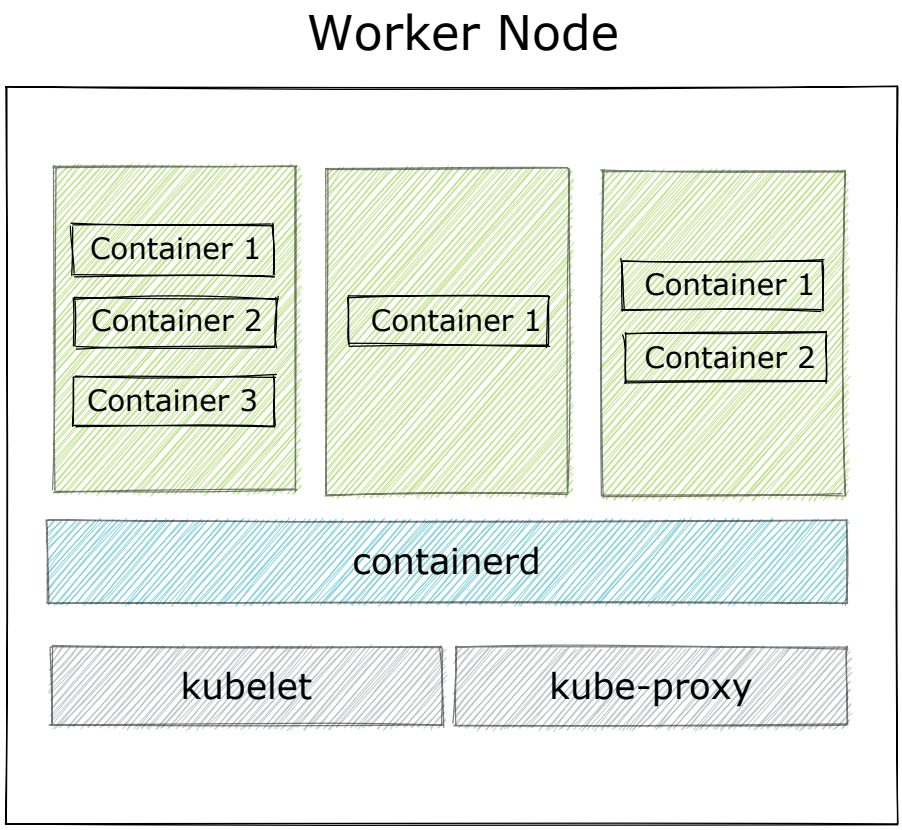

Figure 2.2: Kubernetes worker node with pods

Pods

Worker nodes in a Kubernetes cluster host Pods, and Pods tis the smallest deployable unit of computation in Kubernetes [70]. A Pod is a group of one or more containers with shared resources and configuration. The containers within a Pod are typically tightly-bound forming one *logical* application. An example of an additional container in a pod could be initialization containers that perform work before an application is started.

Pod Implementation Pods are a logical component in a Kubernetes cluster. On the Nodes in the cluster pods are realised through Linux namespaces, cgroups, and potentially other kinds of isolation. Figure 2.2 shows an example of a worker node in a Kubernetes cluster with three pods, note that the three nodes have a different amount of containers making up the Pod.

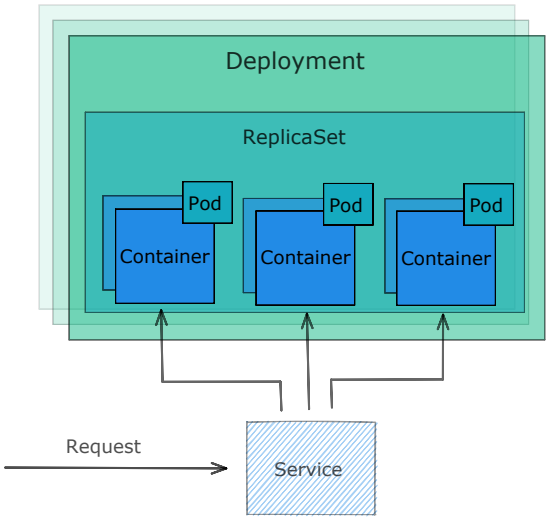

Figure 2.3: Fundamental Kubernetes concepts

Sidecars A common pattern is to extend and enhance containers in pods with "sidecar containers". Sidecar containers are usually used to provide non-business logic features, meaning if the feature is custom or tied hard to the business function the application or service is trying to solve it should probably not be provided as a sidecar container. This leaves use-cases such as monitoring, access control, logging etc. For example Google has created a proxy for accessing databases in Google cloud that uses the sidecar pattern [38]. But most importantly this is the pattern that *service meshes* use to implement their data plane. See 2.6 for more.

Services

A Kubernetes *Service* is an abstraction for a grouping of pods as a network service. The Service main function is to provide a stable DNS name for the group of pods. Pods are non-permanent resources, they can be created and destroyed depending on *Deployments*. A Service provides a stable endpoint and can load balance over the underlying pods.

ReplicaSets

A ReplicaSet defines how many replicas of a pod should be instantiated in the Kubernetes cluster. ReplicaSets is Kubernetes' way of scaling applications horizontally. From an end-user perspective one only has to define the desired number of replicas and Kubernetes will make sure that that number of pods are running at any given time in an eventual consistent manner.

Deployments

The Deployment is where the end-user defines such configurations like ReplicaSet and Services. Deployments describe the desired state and Kubernetes changes the cluster state to match in a controlled roll out.

Extending Kubernetes

Kubernetes has been described by many as the 'new operating system of the cloud' [43]. Kubernetes extensible architecture, rich with extensions and composable parts allows organisations to build custom internal platforms. A key part of Kubernetes that allows this extensibility is *Custom Resources* and *Custom Controllers* that work upon them. This is the build blocks that ex. *service mesh* implementations, see 2.6, used to run the data plane, control plane and the configuration that governs the mesh.

Custom Resources Resources are endpoints in the Kubernetes API [72]. In Kubernetes they are backed by the *etcd* data store. The Kubernetes API stores a collection of different kinds of resource objects. Similar to Class and objects in an Object-oriented programming language. Kubernetes contains several built-in resources like *Pods* and the Kubernetes API contains a collection of *Pod* objects. *Custom resources* are extensions of the Kubernetes API, resources not available in a default Kubernetes cluster. More and more functionality in Kubernetes is delivered through *custom resources*. *Custom resources* are dynamically registered, created and destroyed during cluster runtime.

Custom Controller *Custom resources* and the Kubernetes API lets user complete CRUD operations with structured data in the Kubernetes cluster. To make a true *declarative API custom controllers* are needed. Declarative API is core to Kubernetes, as the user states the intended state of the cluster and Kubernetes works to make the cluster state match. *Controllers* work on resources in the cluster continuously working to keep desired and true state in sync.

Operator Pattern *Custom resources* combined with *custom controllers* is key to implement the *operator pattern* [69] commonly used in Kubernetes. Using the *operator pattern* in Kubernetes, administrators can automate custom behaviour like application deployment, backups, integration with systems outside of the cluster. The *operator pattern* is similar to control loops. In robotics and automation, a control loop is a non-terminating loop that regulates the state of a system.

2.4 DevOps

DevOps is the practice of organising for collaboration between developers and operations. In traditional enterprise developers have developed software and

"thrown it over the wall" to operations that are charged with running it. This way of producing projects and products is highly antagonistic where developers are incentivized to develop new features and for putting them into production fast, and operations are incentivized to keep the systems stable and reliable. When developers and operations are pitted against each other by way of their incentives the whole organisation suffers, leading to loss of morale, burn-out, and reduced innovation [66].

2.4.1 Core Principles of DevOps

- Maximize flow of work from development to customers through operations. Doing this by making work visible, reducing batch size and intervals of work, building in quality and optimizing for overall organisation goals. Practices that lead from this are; continuous build, integration, testing and deployment, creation of environments on=demand, and building software and processes that are safe to change.
- Enable fast and consistent flow of feedback in all stages of development. This enables faster failure detection and recovery. Seeing problems as they occur and engage all necessary personnel in swarming on problems.
- Work to create a high-trust culture that supports experimentation, risk-taking and organisational learning from success and failures. Additionally work to transform local discoveries into organisation wide improvements, sharing knowledge and experience.

Higher levels of DevOps evolution mean more self-service offerings for developers. Highly evolved firms offer a wide range of self-service capabilities, including:

- CI/CD workflows
- Internal infrastructure
- Public cloud infrastructure
- Development environments
- Monitoring and alerting
- Deployment patterns
- Database provisioning
- Audit logging

In then recent *State of DevOps Report* [127] 63 percent of the surveyed companies said they were using internal platforms to develop and run software. The idea is that internal *platform team(s)* provides a platform for *product or projects teams as-a-Service.*

2.4.2 Team Topologies

An evolution in the DevOps ecosystem has been the development of the *Team Topologies* [142] model. The Team Toplogies model tries to model how an organisation should structure their teams and responsibilities for fast flow. The model builds upon existing ideas in the space such as Conways law [24], API-first [74], and DevOps. The model includes several ideas on how to best organise for fast flow, but at the core is a structured way to look at teams on how they serve the organisation.

Four Fundamental Topologies

- **Stream-aligned team:** Aligned to a flow of work from (usually) a segment of the business domain. Also commonly referred to as product teams.
- **Enabling team:** Helps a Stream-aligned team to overcome obstacles. Also detects missing capabilities.
- **Complicated Subsystem team:** Where significant mathematics/calculation/technical expertise is needed.
- **Platform team:** A grouping of other team types that provide a compelling internal product to accelerate delivery by Stream-aligned teams. The role of a developer in a platform team is similar to *Site Reliability Engineer* [7], which is Google's approach to DevOps.

2.5 Microservices

The microservice architecture popularized by organizations such as Google, Netflix and Amazon. The architecture draws from other sources such as Domain-Driven Design, DevOps, CI/CD, cloud computing, REST and maybe most importantly UNIX [110]. Microservices are small autonomous services that work together to make up an application. Each microservices is its own process running separate from the other services. Microservices communicate over networked APIs commonly HTTP. There is no official standard for microservice architecture, but a microservice architecture commonly display the following characteristics [76]:

- Componentizing software is by breaking down into service
- Microservices are independently deployable
- Microservices are organized around business capabilities
- Decentralized Governance, microservices can be built using different programming languages and technology stacks
- Smart endpoints and dumb pipes, microservices communicate over simple protocols like HTTP or similar

- Decentralized Data Management, microservices control their own data store and exposes data over defined APIs

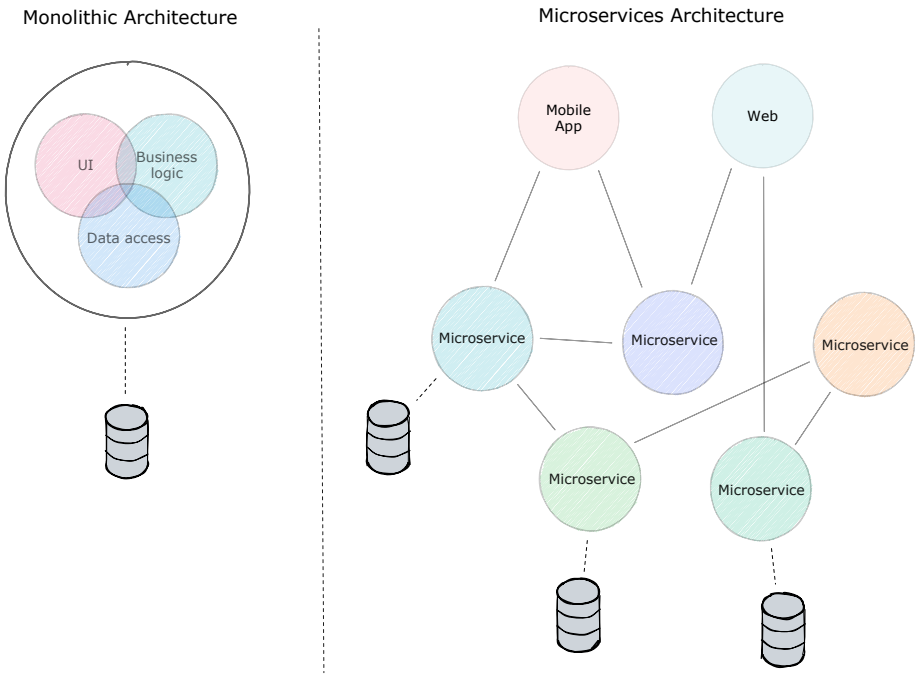

Figure 2.4: Monolith vs microservices architecture

Developer teams favor microservice architecture, especially in large organisations because it provides a technical barrier between teams so they can develop their services independently, thereby increasing development speed and providing the necessary space for innovation. But speed and technical separation over network boundaries does not come without cost, Christian Posta, author of *Microservices for Java developers* notes that :

> *As we move to services architecture we push complexity to the space between our services* (Source: [125])

When calling APIs in a microservice architecture you might believe that the call will be directly from service A to service B, but that is not how networks operate. When integrating services over the network you are subject to the network's performance, reliability, security and faults. In addition to service discovery, finding the service you want to connect to? Load balancing, how do we distribute load over our deployed microservices? Service mesh technology was developed as an answer to this need in organisations utilizing a microservice-oriented architecture, see 2.6. Figure 2.4 shows a simplified comparison between

microservices and a monolithic architecture, note that in a traditional monolithic architecture different responsibilities are all bundled together into one artifact, usually backed by a centralized data store. With microservices the full system is delivered through many different services doing their part, this enables decoupling of things like the user interface so web applications and mobile apps can use the same backing services, additionally the different services can use the type of data store best suited for their purpose.

2.6 Service Mesh

A service mesh is a relative technology space, first emerging in the cloud native ecosystem with Linkerd in 2016 [99]. Service Mesh were described like this in a review paper from 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)

> *A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.* (Source: [78])

As mentioned in section 2.3 and 2.5 the architecture patterns and technologies that are being used to develop software are changing. While the new ways of developing software have many benefits there are serious drawbacks as well. Primarily the drawbacks from creating distributed systems. Engineers wanting to create distributed systems need to deal with the *The 8 Fallacies of Distributed Computing* [123].

2.6.1 Service Mesh Requirements

The natural inclination for developers might be to implement the handling of retries, service discovery, security, etc, in the application or service code. This might be a good solution for one application, but when there is a fleet for services a more reproducible approach is suitable. So the shift to microservices means new requirements that need to be handled. Some of the requirements that need to be handled are:

1. Observability
2. Reliability
3. Security
4. Discovery

2.6.2 History of Service Mesh

The first iteration of what would eventually become service mesh started with the early pioneers in the microservice space. Netflix [109], Twitter [163] and Facebook [31] all created reusable libraries that application and service developers could use to connect safely and reliably to other services in their respective organisations. While libraries are good, they allow for reuse, they are also limited to the programming language or platform which they are implemented in. This meant the organisations that implemented these libraries needed to standardize on one platform. Additionally releasing a new version is problematic, requiring all users of the library to upgrade in tandem. The natural progression was to move the libraries that powered the first service meshes into independent proxies that could be configured and updated outside of the applications and services that rely on it. Buoyant where the first to do this by wrapping the Twitter project Finagle [163] into a service mesh offering named Linkerd, first released in 2016 [99], later others followed and the landscape is filled with different service meshes, mainly open-source, competing for market share. Figure 2.5 how the service mesh moved from in-process as libraries, to out-of-process as proxies.

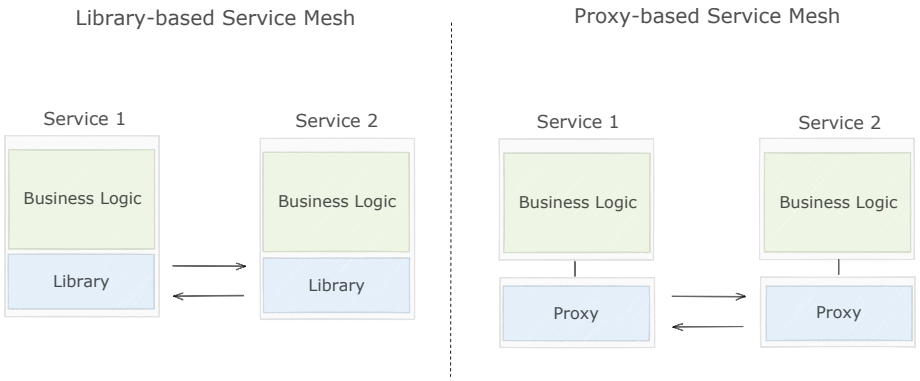

Figure 2.5: Library vs proxy-based service mesh

2.6.3 Service Mesh Architecture

The basic architecture of a service mesh consists of two components; the data plane and the control plane. Figure 2.6 presents a generic example.

Control Plane

The Control plane in a service mesh manages and configures the data plane. The control plane also typically exposes APIs or interfaces for users to observe and manage the service mesh. Typical configuration options are; access policies,

routing decisions, and rate limits. Examples of popular service mesh data planes are Istio [55] and Linkerd [82].

Data Plane

The data plane is responsible for handling the request in and out of the upstream service. The data plane is transparent to the upstream service, meaning no changes to the service code need to be implemented to use the data plane. The data plane consists of proxies managed by the control plane. In Kubernetes such proxies are deployed to Pods in the sidecar pattern as described in section 2.3.3. Examples of proxies used in popular service mesh implementations are Envoy [27] and linkerd2-proxy [81].

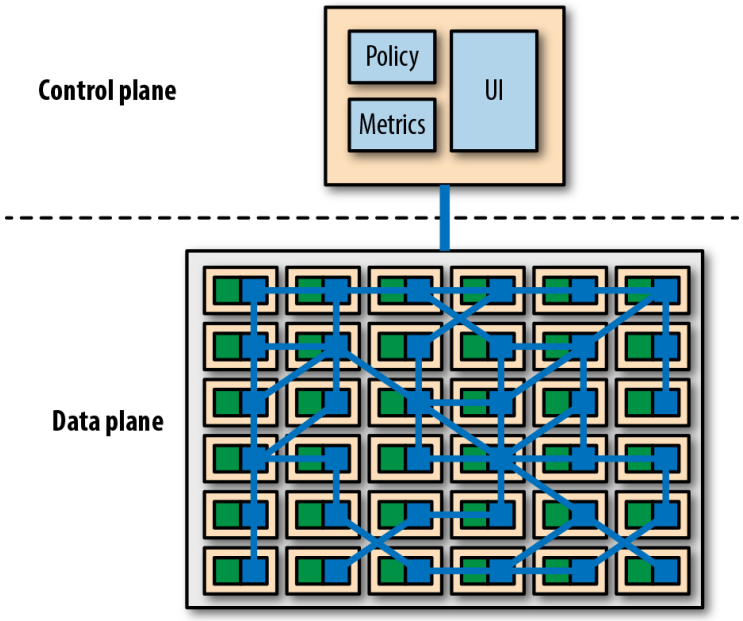

Figure 2.6: Basic service mesh architecture (Source: [98])

2.6.4 Common Service Mesh Features

- **Observability:** The service mesh, through the data plane proxies, are able to record and share fine grained metrics about both layer 4 and layer 7 level events, like TCP reconnects or HTTP 500 responses. Providing a complete, service independent view of all traffic in the mesh. In addition service meshes can provide *distributed tracing* like *Jaeger* [152].
- **Reliability:** The service mesh can manage failed requests in a graceful manner. Service meshes can understand layer 7 protocols like HTTP

and can act on metadata like status codes. Service meshes can typically provide timeouts, retries, deadlines, load balancing and circuit braking.

- **Security:** The data plane introduces a consistent and fine grained injection point for service authentication and access control. Traffic between proxies can be encrypted, see 2.6 for more on the security capabilities of service mesh.
- **Discovery:** Service mesh has control over all active services in the mesh and where they are located. In the transient world of microservices service meshes can route based on real-time data about the services.

2.7 Envoy Proxy

Envoy is a high performance proxy built in C++ design as a proxy for single applications, and large service mesh architectures [26]. Envoy was originally developed at Lyft as a reverse proxy for Lyft's growing web of interconnected microservices. Envoy is the data layer proxy for most service mesh offerings. Envoy advertises the following key features:

- Out of process architecture
- HTTP/2 ad GRPC support
- Advanced load balancing
- TLS termination
- Circuit breakers
- Health checks
- Staged rollouts with %-based traffic split
- Fault injection
- API for configuration management
- Observability

2.7.1 Envoy Performance

Performance is extremely important for proxies and Envoy is no exception. Proxy performance is a tough problem as the use cases are highly diverse. In a recent benchmark of five of the most popular proxy services Envoy showcased just how leading it is in the space [18], see Figure 2.7 which shows HTTPS request per second performance scores.

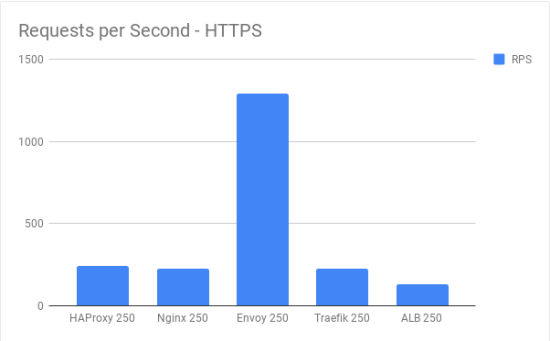

Figure 2.7: Chart of Requests per Second over HTTPS by Load Balancer and Concurrency Level (Source: [18])

2.7.2 Envoy High Level Architecture

Envy abstracts the network away from the services and applications it supports. Inside the Envoy processing of a request is divided in two mainly sequential parts. See figure 2.8 for an overview of the main parts in the system. A main thread handles server lifecycle, configuration and administration features like metrics. Work in Envoy is distributed over worker threads that each contain a *Listener* and *Cluster* subsystem, bridged by the *HTTP route filter*. The full lifecycle of a downstream connection is handled by one worker thread [30].

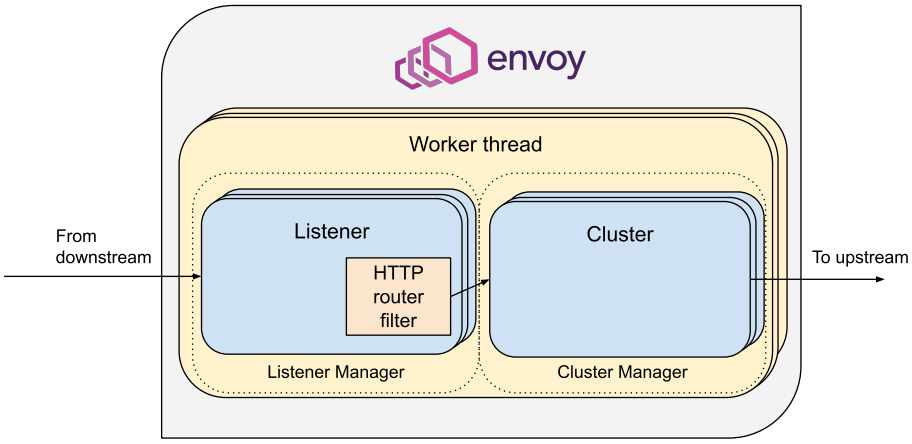

Figure 2.8: Envoy High Level Architecture (Source: [30])

- **Listener subsystem** handles processing of requests from downstream, manages the whole lifecycle of the incoming request to outgoing response.
- **Cluster subsystem** responsible for upstream connections, including up-

stream endpoint health, load balancing and connection pooling.

2.7.3 Envoy Extension

The listener subsystem exposes an interface for extensions, called filters, through static compiled extensions written in C/C++. Requiring all extensions to be compiled in at build-time. Projects like Istio therefore needs to create, link, build and release their own versions of the Envoy proxy [53]. Envoy does come with an array of built-in filters that are available by default [27], but most service mesh projects need additional extension for their feature set. The built-in filters include caching, tracing, CORS, TLS, etc.

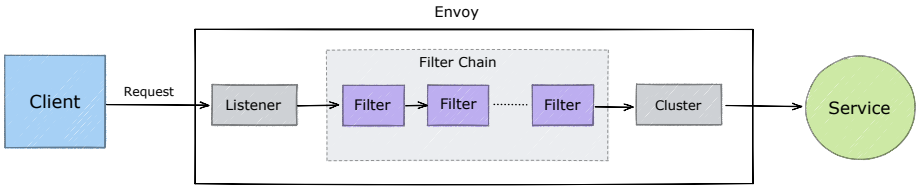

Figure 2.9: Simplified diagram of request processing in Envoy

Envoy Filter Architecture

Figure 2.9 shows a simplified view of how a request is processed in Envoy and how filters, built-in and custom are applied to the request. Filters can be basic network filter, or more specialized HTTP filters. When a new downstream request reaches the listener a filter chain based on configuration is created to process the request. As figure 2.9 shows the filters are composed into a sequence and are processed in order. The model is event based, each filter is called on an event in the request lifecycle, like *on_http_request_headers*. Filters can stop and continue iteration to subsequent filters. The filters can be applied to the request, upstream response or both. The last filter that runs is the *HTTP route filter* [29] that routes the request to the correct cluster.

2.7.4 Extending Envoy with WebAssembly

Extending Envoy with compile time filters is powerful and has been used to success for many projects. However it is not without problems, first it is not easy to setup and maintain a custom build pipeline for Envoy, extensions implementation is restricted to C/C++, and updating an extensions requires a full build, test, delivery process [22]. Several service mesh projects have begun to create their own mechanism to extend their service mesh, making it possible for users to have a more developer friendly mechanism of developing and deploying custom extensions to their service mesh. This approach is also not unproblematic, every new extension system is service mesh specific and fragments the ecosystem. WebAssembly extensions were introduced to Envoy to

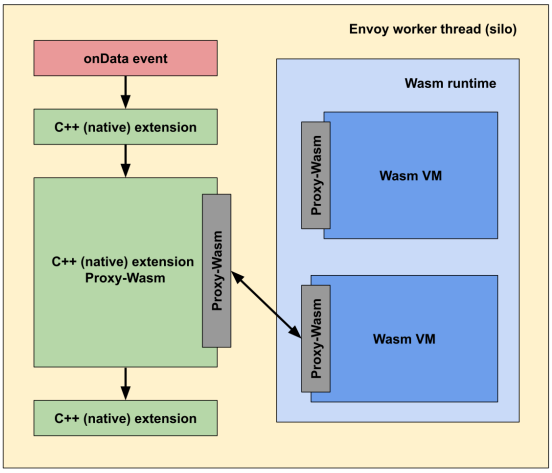

Figure 2.10: Diagram showing how the Envoy proxy interacts with WebAssembly filter modules (Source: [139])

solve this problem by letting developers extend Envoy with modules compiled to WebAssembly that can integrate with the filter chain like a native filter. Figure 2.10 shows how a WebAssembly VM is integrated into the filter chain through an *Application Binary Interface (ABI)*. Using WebAssembly allows developers to develop extensions using many different languages and leverages the full WebAssembly ecosystem for tooling and support. See section 2.8 for a more in-depth description of WebAssembly.

2.7.5 WebAssembly Proxy Specification

To make WebAssembly proxy extensions reusable across different proxies work is being undertaken to standardize on a specification for WebAssembly proxy extensions named *Proxy-Wasm* [140]. The ABI specification includes many imports and exports for implementing modules for proxies [140]. Including:

- Managing contexts
- Bytestream events
- HTTP events
- Grpc events
- Logging
- Host supported key-value store

- Host supported queue
- Host supported timers
- Metrics
- Interface for foreign functions

Design

The specification defines an Event-driven streaming ABI. The implementer for modules that uses the specification create implementation of handler functions that are called by the host environment, the proxy, when a specific event happens.

```
1 pub extern "C" fn proxy_on_request_headers(context_id: u32, num_headers: usize) -> i32
```

Source Code 2.1: Implementation of event handler from proxy-spec, exported from wasm module to host

Context Each request that is filtered by a WebAssembly module has a *Context* created instantiated for handling it during the lifetime of the request. *context_id* is passed with the event call to the module from the host, and the module is responsible for managing any internal state for a specific context id. This management is made easier by the SDKs created for the supported programming languages [138].

Example

Example 2.2 is pseudo code of the event handler that is called for each chunk of HTTP request body for the context *context_id* received from downstream. The actual body bytes can be retrieved from the host by using *proxy_get_buffer*, also described in the specification. The return value *action* instructs the host environment what actions are to be taken next.

```
proxy_on_http_request_body(context_id, body_size, end_of_stream) -> action
```

Source Code 2.2: On HTTP request body event handler

Actions The actions the filter can return to the host environment, and into the Envoy filter chain process.

- **Forward:** The HTTP request body chunk should be forwarded upstream.
- **EndStream:** The HTTP request body chunk should be forwarded upstream and the stream should be closed afterwards.

- **Done:** The HTTP request body should be forwarded upstream and this extension should not be called again for this HTTP request.
- **Pause:** Means no further processing of the request until the *proxy_ http_ resume_ request* is called.
- **WaitForMoreData:** Means that further processing should be paused and host should call this event handler again when it has more data.
- **WaitForEndOrFull:** Means that further processing should be paused and the host should call this event handler again when the complete HTTP request body is available or when internal buffers are full.
- **Close:** Means the stream should be closed immediately.

2.8 WebAssembly

> WebAssembly: it's neither web, nor assembly
>
> *Ancient wasm dev proverb*

WebAssembly, or *WASM* as it is commonly referred to as, is a portable bytecode instruction format for a stack-based virtual machine, created as an open standard inside the W3C WebAssembly Community Group [170]. The WebAssembly is a target for compilation for code written in multiple languages. It advertises near-native speed of execution [169]. Several programming languages now support compilation to WebAssembly [4], but at time of writing this thesis C, C++ and Rust has the best support all-around.

As a platform for software development WebAssembly consists two parts:

- A Virtual machine that can run WebAssembly modules, like Google's high-performance JavaScript and WebAssembly engine, V8 [161], or Wasmtime [11], a standalone WebAssembly runtime.
- API like the WASM Web API [172] for modules to be ran in a browser environment, and WASI [171] for both browser and server side environments.

> *It's a tsunami that can change not only how consumers interact with and how developers build applications but also fundamentally alter the kinds of applications we can create. It may even transform our core definition of the word application.*
>
> *- Kevin Hoffman, Programming WebAssembly with Rust (2019)* [47]

2.8.1 History and Use Cases

Javascript has some problems. Including Javascript in a web page means the user-agent needs to download, tokenize and parse the Javascript to execute the code. asm.js was a project to try to make Javascript faster by creating an optimized subset of Javascript developers could *transpile* Javascript code to [44].

WebAssembly would provide performance gains for higher-level languages like Python or Ruby and security through WebAssembly's sandboxing for lower-level languages.

2.8.2 WebAssembly Architecture

Unlike the typical machines people generally use today, which are *register machines*, the WebAssembly instruction format is for a *stack-based machine*. Stack-based machines have some advantages over register-based ones: Small binary size, efficient instruction coding, and ease of portability [47]. Other well know stack-based virtual machines are the *Java Virtual Machine (JVM)* [79] and *Common Language Runtime (CLR)* [91]

The WebAssembly encodes a simple low-level language that provides basic primitive building blocks that are designed to be secure, compact, and performant. The language includes basic value types, instructions, traps, functions, tables and linear memory. The WebAssembly virtual machine has no concept of heap like other machine languages usually has. It instead has something called *linear memory*, a continuous block of bytes that can be declared internally in a WebAssembly module, exported out of the module, or imported from the host. Again the benefit is security, the host can read and write any linear memory given to a WebAssembly module, but the WebAssembly module can never access host memory directly [164]. Additionally the WebAssembly language defines two important concepts:

- **Module**, a WebAssembly binary that contains definitions for functions. tables, linear memory, and global variables. Modules are the *embedded* into a host environment.
- **Embedder**, responsible for *embedding* a module into a host environment. Responsible for loading modules, defining how imports are provided, and how a module's functions are exported.

WebAssembly Module Figure 2.3 presents an example of a small WebAssembly module encoded in the WebAssembly Text Format (wat) [87]. Wat is a human readable format designed to help developers view the source of an WebAssembly module. The text format can also be used for writing code that can be compiled to the binary format. Note that the example module start with an import statement that imports a "sys.print" function from the host environment. Following the import the module defines a block of memory the module exports to make available to the host environment (like a browser). Finally the

module exports a "main" function to the host environment where the module uses the imported function with a constant as an argument.

```
(module
  (func $import0 (import "sys" "print") (param i32))
  (memory $memory0 200 200)
  (export "memory" (memory $memory0))
  (export "main" (func $func1))
  (func $func1
    i32.const 0
    call $import0
  )
  (data (i32.const 0) "Hello, world\00")
)
```

Source Code 2.3: Wat formatted WebAssembly Module Example

2.8.3 WebAssembly Security

One of the big benefits of using WebAssembly as a compilation target and runtime of applications, both in the browser and server-side, is security. The WebAssembly security model is designed to protect against potential malicious modules and support development by providing secure primitives. Modules run in a sandboxed environment and can only interact outside of the sandbox though imported functions. Out of the box WebAssembly does not provide any access to the computing environment a WebAssembly module is running, like system calls [141] usually provided by operating systems, meaning a module cannot do things like open files or write to terminal arbitrarily [165]. Any and all interactions with the environment has to be done through functions provided by the embedder. The security model limits the damage a malicious third-party dependency can do unlike other popular languages for application development [113], making WebAssembly suitable for security critical environments like browsers and proxies.

WebAssembly Security Features [159]

Host Environment

- WebAssembly modules execute in a sandboxed environment.
- WebAssembly modules execute deterministically, meaning no undefined behaviour in the langauge semantics (with only a few exceptions [158])
- WebAssembly modules are also subject to the embedder's security policies.

Control Flow The design of WebAssembly promotes safe programs by eliminating dangerous features from its execution semantics.

- Since compiled code is immutable and not observable at runtime, WebAssembly programs are protected from control flow hijacking attacks [122].
- A protected call stack that is invulnerable to buffer overflows in the module heap ensures safe function returns.
- WebAssembly supports implicit enforcement of control-flow integrity (CFI) through structured control-flow [2].
- Traps are used to immediately terminate execution and signal abnormal behavior which immediately aborts execution. Traps cannot be handled by WebAssembly code, but are reported to the outside environment, where they typically can be caught.

Memory Safety WebAssembly language semantics obviate certain classes of memory safety bugs found in other languages.

- Buffer overflows, which occur when data exceeds the boundaries of an object and accesses adjacent memory regions, cannot affect local or global variables stored in index space in WebAssembly.
- WebAssembly enforcement of control-flow integrity and protected call stacks prevents direct code injection attacks. Thus, common mitigations such as data execution prevention (DEP) [94] and stack smashing protection (SSP) [137] are not needed by WebAssembly programs.
- No null-pointers. In WebAssembly, the semantics of pointers have been eliminated for function calls and variables with fixed static scope.
- While return-oriented programming (ROP) [144] attacks are possible, conventional attacks using short sequences of instructions ("gadgets") are not possible in WebAssembly, because control-flow integrity ensures that call targets are valid functions declared at load time.

2.8.4 WebAssembly System Interface(WASI)

WebAssembly because of its stated benefits is increasingly being used beyond the browser, but to create useful software that provide value a interface to build upon needs to exist, like POSIX [166] for UNIX systems and win32 for Windows platforms [96]. WASI is the leading project for creating such a system interface for the WebAssembly platform. WASI is a modular set of standards [171]. Starting with the wasi-core standard, wasi-core provides a similar interface to POSIX. A core idea behind WASI is capability-based security [100], where access to external resources is always represented by *handles*. The way this works is that the only way to obtain access to resources through the WASI API is to be

given handles, or perform operations on handles to receive new handles [46]. The capability-based security model provides suitably fine-grained access control and an intuitive model for developers using WASI to create modules.

> *If WASM+WASI existed in 2008, we wouldn't have needed to create Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!*
>
> - *Solomon Hykes, Co-Founder Docker inc.* (Source: [49])

2.9 OAuth 2.0

OAuth 2.0 is an authorization framework that enables applications to obtain limited access to other HTTP services on behalf of a resource owner. There exists a lot of confusion about what OAuth is, and what it delivers. The correct way to think about OAuth is that it supports *delegated authorization*, Usually this means an end-user delegates authorization for another service to an application. OAuth allows applications to get limited access to user data stored in other services without exposing user credentials to the application [129]. So the authorization is not for end-users access, it is what an end-user authorizes an application to access. There exists an OAuth 1.0 specification, but OAuth 2.0 is in no way backwards compatible to OAuth 1.0, and represents to a large degree an independent framework [16]. OAuth is often called a "protocol for protocols" [147], meaning it is used as a basis for other specifications like *OpenID Connect* which is described in section 2.10. For newcomers to OAuth the framework can be hard to understand completely as the full specification used by *authorization server* providers it is not defined by one RFC(Request for Comments) [51], but several RFC documents that define and redefine certain aspects of the framework. Providers of OAuth services do not always support the same set of standards.

2.9.1 Before OAuth 2.0

Before the adoption of OAuth, developers wanting to develop clients, applications integrating with resource servers, usually requested that users provide their credentials to the resource server. This had several security and user experience related issues for users [16]. OAuth delivers a more secure way to integrate services and provide a standardized platform for integration.

- Clients would have to store users passwords for resource servers, increasing impact in case of a breach.
- Resource servers need to support password authentication, despite the weakness of password authentication.
- End-users are unable to restrict client access to resource servers.

- End-users cannot revoke access to one client without revoking access for all clients given access to a resource server, and must do so by changing the end-users password to the resource server.

2.9.2 OAuth 2.0 Definitions

The OAuth 2.0 specification introduces some key definitions.

Resource owner An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

Resource server The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

Client An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).

Authorization server The server issues access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

Access Token Access Tokens are credentials, usually used by clients, to access protected resources. Tokens represent specific scopes and duration of access, granted by the resource owner, usually an end-user. The authorization server is responsible for issuing valid tokens and the resource server is responsible for validating tokens before allowing access to protected resources. Access token, as by the OAuth 2.0 specification [16], can have different formats. structures, and cryptographic properties depending on the resources servers security requirements, but today JSON Web Tokens [60] are the most used token type for access tokens.

Refresh Token Refresh tokens can optionally be returned for an authorization server to a client, usually alongside the access token. The refresh token enables the client to request new access tokens when the previous becomes invalid or expires. The refresh token is usually an opaque string with no encoded meaning in the token itself.

2.9.3 Authorization Grant types

The original OAuth specification [16] includes several Authorization Grant types, but the latest RFC regarding OAuth best practice recommend only the use of Authorization Code or Client Credentials [42].

Authorization Code

In the Authorization Code flow the authorization server serves as an intermediary between the client and the resource owner. The client receiving an not-authorized request, redirects the end-user, via the end-users user-agent [75], to the authorization server. After authenticating the end-user and obtaining authorization for the client, the authorization server redirects the end-user back to the client with the authorization code. The client, after receiving the authorization code through the redirect can, without passing through the end-user, exchanges the authorization code for an access token with the authorization server. The security benefits of this flow is that the client can be authenticated, the resource owner's credentials are never shared with the client, and the access token is never passed through the end-user removing the possibility of exposure. The latest recommendation is to extend Authorization Code flow with PKCE(Proof Key for Code Exchange) [105] where an additional code verifier is used to protect against malicious intercept of the authorization code.

Implicit

Implicit flow is a simplified authorization code flow. Meant for clients implemented in the browser, such as Javascript applications. In the implicit flow, instead of receiving an authorization token, the client receives an access token directly. This flow has some security disadvantages compared to Authorization Code flow, the client is not authenticated, and the access token is exposed to the resource owner and user-agent.

Resource Owner Password Credentials

The resource owner's credentials, password and username, can be used directly by the client to obtain an access token from the authorization server. The OAuth 2.0 specification recommends this flow to only be used when there is a high level of trust between the end-user and the client, like an operating system or highly privileged application [16].

Client Credentials

The client can use its own credentials to retrieve an access token from the authorization server. Usually this flow is only used when the authorization scope is limited to resources under control of the client, or authorization to the resources as has been previously established.

2.9.4 Access Token types

The access token returned after a successful token request to the authorization server is returned with the type of token that is returned. The type property provides the client with sufficient information on how to use the token successfully when calling a resource server.

Bearer Token A *bearer token* is an access token type with the property that any holder of the token, the bearer, can use the token in any way any other holder could. In short this means that the bearer token can be used without proving possession of any other cryptographic keys or secrets. In industry JWT(JSON Web Tokens) [60], see section **??** are usually used to encode the bearer token which includes digital signatures through a JWS(JSON Web Signature) [59], see section **??**.

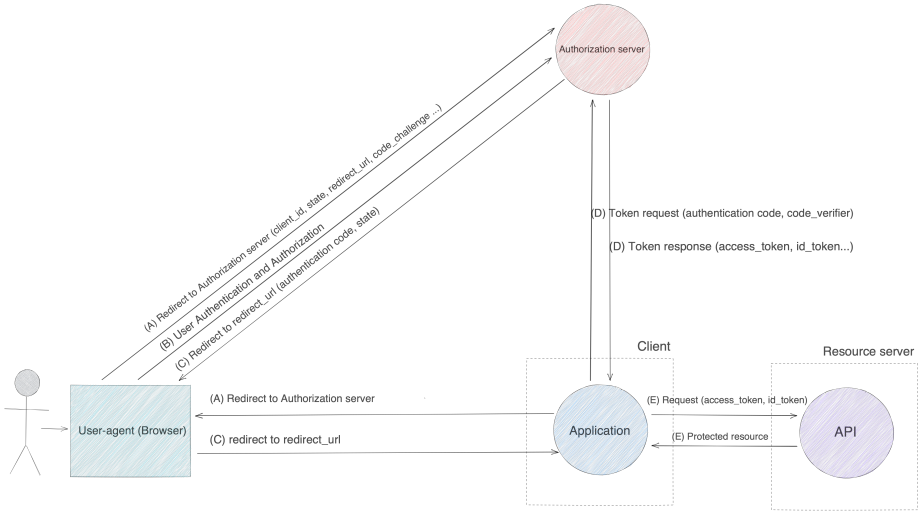

Figure 2.11: Authorization Code flow with OAuth 2.0

2.9.5 Delegated Authorization with OAuth 2.0

Example of a delegated authorization using OAuth 2.0 Authorization Code grant flow with PKCE. Figure 2.11 presents how the flow begins with the user, displayed by the stick figure, through the different actors in the specification to finally successful access to the protected resource.

(A) The Client initiates the flow by redirecting the Resource Owner (end user) to the authorization server endpoint. Included in the redirect is *state*, *client identifier*, *requested scope*, *redirection URI*, and *code challenge*. The redirection URI tells the authorization server where it should redirect the Resource Owner to after authentication and authorization is completed (or denied). The code challenge is a hashed value of the *code verifier* the client stores for the token request as part of the PKCE extension [103] .

(B) The Authorization Server authenticates the Resource Owner and establishes whether the Resource Owner grants or denies authorization based on the requested scope from the Client.

(C) Assuming the Resource Owner grants the access, the Authorization Server redirects back to the Client using the redirection URI provided in the initial redirect, or predefined during client registration. The redirection URI to the client includes an authorization code and the state parameter from the initial redirect (CSRF protection).

(D) On receiving the redirect from the Authorization Server the Client requests an *access token* from the Authorization Servers token endpoint. The client includes the authorization code from the previous step, the redirect URI used in step (C), and the code verifier matching the code challenge included in the initial redirect. The Authorization Server authenticates the Client request, validates the authorization code and matches the code verifier with the code challenge it received in the initial redirect, and ensures that the redirect URI included in the request matches the redirect URI used in step (C). If valid the Authorization Server responds back with an access token, and optionally a refresh token.

(E) Receiving the access token the Client can now use the access token to request protected resources from the Resource Server, in this case an API. The Resource Server will validate the token either by interacting directly with the Authorization Server using a token validation endpoint, or if the token is a JWT the token can be validated using JWT validation [60].

2.9.6 OAuth 2.0 Vulnerabilities

The OAuth 2.0 framework and the protocols built on-top of it has not been without its fair share for specification and implementation related vulnerabilities [124]. OAuth have many moving pieces and optional components. Some common client implementation flaws that are relevant for this thesis are:

- Flawed CSRF protection
- Flawed scope validation
- Unverified user registration (mismatch between what level the client assumes the user has been authenticated and what the authorization server actually has performed).

2.10 OpenID Connect

OpenID Connect, or as the specification refers to it as OpenID Connect 1.0, is an identity extension to the OAuth 2.0 framework [133]. It extends OAuth 2.0 with the ability to verify the identity of end-users based on the authentication performed by the authorization server. OpenID Connect additionally also includes specification on how a client can obtain basic profile information from an authorization server on a REST [33] endpoint. Figure 2.12 shows how the OpenID Connect protocol is built from several OpenID, OAuth and auxiliary specifications.

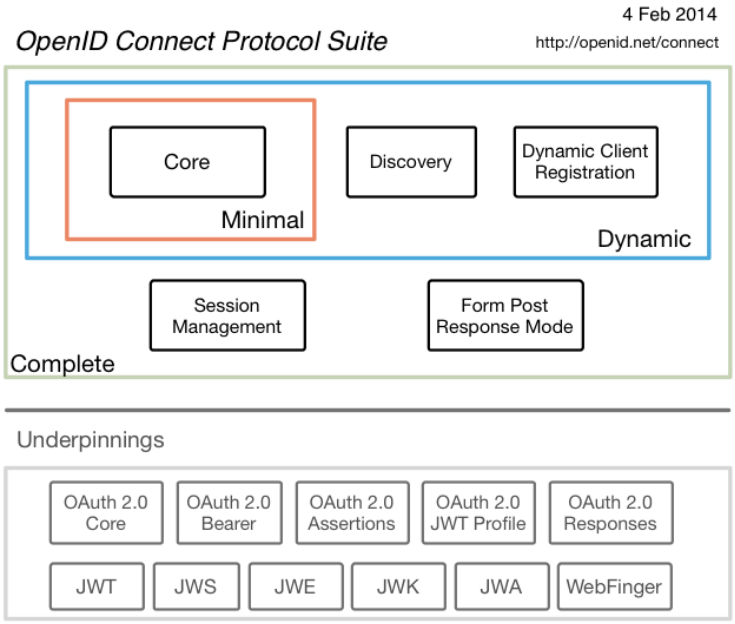

Figure 2.12: Diagram showing the different specifications that OpenID Connect is made up of and builds upon (Source: [118])

2.10.1 OpenID Connect Definitions

Authentication Request

Extension to OAuth 2.0 Authorization request, see 2.9.3, using extension parameters and scopes, defined by the OpenID Connect specification, so that end users be authenticated by the Authorization Server, which is extended to be a OpenID Connect Identity Provider.

ID token

The ID token is the primary artifact in the OpenID extension to OAuth 2.0. The ID token is a security token that contains claims, where claims are pieces of information asserted about an entity [133], about the authentication of an end-user performed by an authorization server. The ID token is encoded as a JWT [60].

2.10.2 OpenID Connect Protocol

OpenID Connect uses the Authentication Request to perform authentication for the Client. The Authorization Request can extend three of the flows in OAuth 2.0, Authorization Code Flow, Implicit Flow and Hybrid Flow. Authorization

Code Flow is intended for Client that can securely maintain a secret and is the performed choice for secure authentication [42].

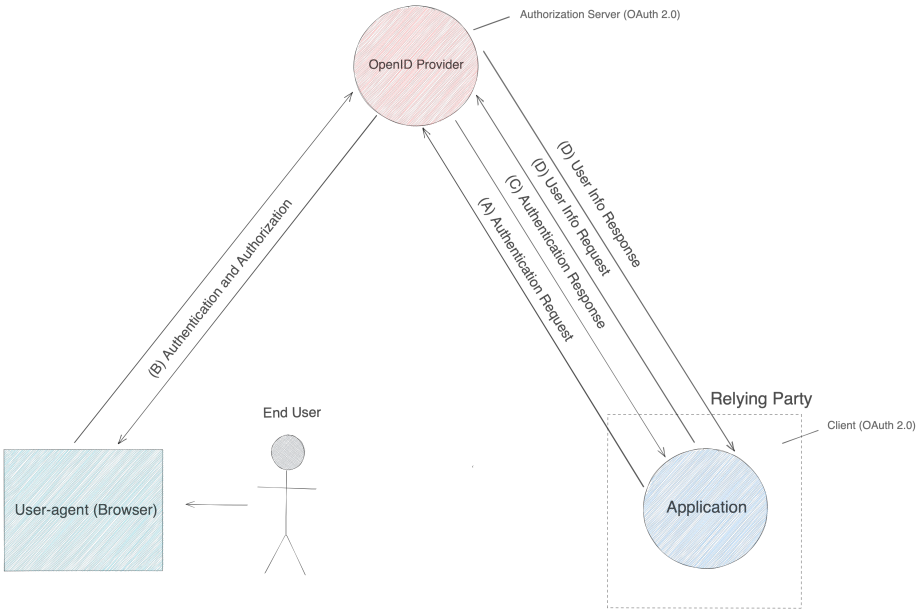

Figure 2.13: Abstract presentation of the OpenID Connect Authentication Request Flow

Authentication Request using Authorization Code Flow

Figure 2.13 presents a simplified and abstract representation of the OpenID Connect protocol. The logical steps are made concrete by mapping to OAuth 2.0 flows.

(A) The Relying Party, or Client using OAuth 2.0 terminology, initiates by sending an Authentication Request to the OpenID Provider, or Authorization Server in OAuth 2.0.

(B) The OpenID Provider authenticates the end user and establishes whether the end user grants or denies giving the Relying Party authorization to access identity information about the end user.

(C) The OpenID Provider, given that the end user grants authorization, responds to the Relying Party with an ID Token and usually an Access Token to the User Info endpoint.

(D) The Relying Party can retrieve additional claims than claims embedded in the ID Token by requesting the User Info endpoint.

2.10.3 Token Validation

The OpenID Connect specification, unlike OAuth 2.0, requires clients to validate the token received back after a successful Authentication Request [132]. The ID token needs to pass all validation criteria, and tokens that have failed validation must in no way be used further. The ID token by specification needs to be a JWT, validation that the token is a well-formed JWT is implicit.

1. The issuer for the OpenID provider must exactly match the *iss* claim.
2. The *client_id* used by the client for the issuer needs to be included in the *aud* claim, or if there are additional unexpected or untrusted audiences.
3. If the ID token has multiple audiences, there should be a *azp* claim present.
4. If an *azp* claim is present, the client should verify that the *client_id* is the claim value.
5. The current time must be before the time in the *exp* claim.
6. The *iat* Claim can be used to reject tokens that were issued too far away from the current time. The acceptable range is Client specific.
7. If the *acr* Claim was requested, the Client should check that the asserted Claim value is appropriate.
8. When *max_age* request is made, the Client should check the *auth_time* Claim value and request re-authentication if it determines too much time has elapsed since the last End User authentication.

2.11 Zero Trust Architecture

When operating on-premise digital infrastructure it is common to utilize perimeter network models[5] with mechanisms such as VPNs(Virtual Private Networks) to provide access from outside of the on-premise network. But when an organisation moves to the cloud a perimeter network model becomes hard to maintain and does not leverage the technical opportunities of cloud. With the shift to the cloud, more and more organisations are experimenting with Zero Trust Architecture Models as described in the book Zero Trust Network[36] and NIST Special Publication (SP) 800-207[107]. An example from industry is Google's BeyondCorp[120].

Perimeter Network The *Perimeter Network Model* is the standard network security model used by large and small organisations all over the world. It involves segregating a network into the following network zones. Figure 2.14 presents an example of a perimeter network architecture.

- **Untrusted** The untrusted network zone, commonly the internet.

- **Trusted** The internal network zone, the network segment or segments commonly used to host employees workstations, internal services etc.
- **Privileged** An internal network zone for sensitive data/traffic. Commonly customer data, transaction records, employee data etc.

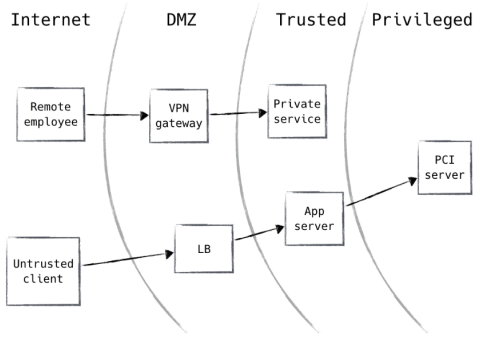

Figure 2.14: Example diagram of a perimeter network (Source: [6])

Zero Trust Architecture also referred to as Zero Trust Networking (ZTN), popularised by John Kindervags report *No More Chewy Centers: Introducing The Zero Trust Model Of Information Security* [67] is at a basic level a simple model; *"no device should be trusted by default"*. Employees work off-site, services are delivered as *Software-as-a-Service*, microservice development, and the increase in network based attacks. In the perimeter model the focus is the network and its boundaries, by protecting the boundaries of the network what resides inside would be safe. The multitude of network breaches that have dotted the Internet ever since the early beginnings are a testament to the weaknesses in this philosophy. When a perimeter network is breached the attacker has free reign, in the book *Zero Trust Networks*[36] the authors refer to perimeter networks as:

> *Secure cells with soft bodies inside*

Zero Trust Architecture places the focus on the actually valuable parts of the network, the resources it contains. Zero Trust Network focuses on protecting the individual resources of the network, not its boundaries. This requires device health attestation, data-level protections, a robust identity architecture, and micro-segmentation to create granular trust zones around an organization's digital resources.

The Zero Trust Architecture Fundamental Assertions A Zero Trust Network is built upon five fundamental assertions[6]:

- The network is always assumed to be hostile.
- External and internal threats exist on the network at all times.
- Network locality is not sufficient for deciding trust in a network.
- Every device, user, and network flow is authenticated and authorized
- Policies must be dynamic and calculated from as many sources of data as possible.

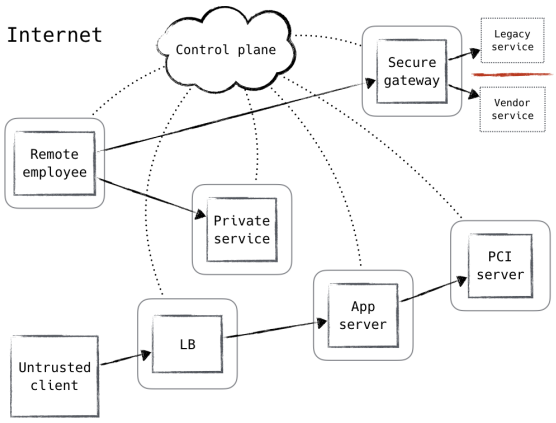

Figure 2.15: Example diagram of a Zero Trust Architecture (Source: [6])

2.11.1 Control and Data plane

The Zero Trust Network is divided into two main distinct parts, the control plane and the data plane. The data plane is the "dumb" layer that directs the traffic on the network. The data plane handles a large amount of traffic so the logic in the data plane nodes has to be simple. The control plane is the brains of the network. The control plane is the entry point for administrators of the network to apply configuration and policies. See figure 2.15 for an example.

2.11.2 Service Mesh and Zero Trust Architecture

Note that the Zero Trust Architecture is very similar to the service mesh architecture described in section 2.6. Several service mesh implementations market themselves as supporting Zero Trust Architecture. A common mechanism is using mTLS between service mesh proxies [40]. mTLS uses x.509 certificates to identify and authenticate each service and a service mesh control plane can issue, update and revoke certificates as needed. mTLS together with access policies governed by the service mesh allows for fine-grained access control between services in the mesh [54] [80].

2.11.3 Token-based Zero Trust Architecture

Token-based Zero Trust Architecture is an application-level approach to Zero Trust Architecture. Token-based Zero Trust Architecture can also be described as *user-centered* Zero Trust Architecture, as it usually relies on using OAuth 2.0 specifications to implement aspects of zero trust [15].

Tokens Security tokens like access tokens implemented using JSON Web Tokens (JWT) allows for scoped access using OAuth 2.0 scopes, see 2.9. Example 2.4 is an example of a JWT with scope and standard claims used for JWT validation. This allows for more fine-grained access control than strictly service-oriented access control like Zero Trust Architecture in service meshes. One example could be that a user grants a service only access to view a user controlled resource, not update or destroy. The access granted by the user can be dynamic and updated based on user actions or policy on the fly [114].

```
{
    "kid":"12",
    "typ":"JWT",
    "alg":"RS256"
}
.
{
    "sub": "my-client-id",
    "aud": "some-api-server",
    "scope": "openid profile",
    "nbf": 1621815729,
    "azp": "mycoolclientid",
    "iss": "http://mock-oauth2-server:8080/customiss",
    "exp": 1621815849,
    "iat": 1621815729,
    "jti": "87eb6917-d12d-405b-9f73-8fc04b1465d0"
}
.
TYugKnWXHmSHhl6xdR7MR7P9rFqlQ4ia... // JWT signature
```

Source Code 2.4: JWT Access Token Example

OAuth Protocols The foundation for using token-based Zero Trust Architecture are several recent OAuth 2.0 extension specifications. Figure 2.16 shows the flow from End User authentication to the subsequent token exchanges the application needs to perform to fulfill the request.

- **OpenID Connect** [118] is fundamental for providing authentication. Robust authentication is a base requirement for any Zero Trust Architecture.

- **OAuth 2.0 Token Exchange** [61] defines a protocol allowing applications to exchange and acquire properly scoped security tokens in order to securely communicate with each other.
- **RFC7523** [62] defines how OAuth 2.0 Clients can craft client assertions, security tokens. Client assertions can be used as client credentials and/or authorization grants used to request access tokens or perform token exchanges. it defines a protocol for using JWTs signed by the application making the token request. Note that it is also possible to use OAuth 2.0 Client Credentials flow (see 2.9) for this purpose, but using JWTs like RFC7523 proposes allows for consistent use of security tokens across the security services.
- **OAuth 2.0 Dynamic Client Registration Protocol** [63] OAuth 2.0 Clients need to be preregistered as authorized OAuth 2.0 clients in order to exchange tokens. OAuth 2.0 Dynamic Client Registration Protocol defines a protocol for dynamically handling client registrations. Client registration is security critical and usually client registration when implemented a trusted party handles the registration, like a strongly hardened platform service. Dynamic client registration is necessary to be able to handle the dynamic nature of cloud native environments where services come and go at a high pace.

Implementations NAV's *Tokendings* [162] service is one example of an open source implementation of token-based Zero Trust Architecture through leveraging service mesh for orchestration and bootstrapping trust through custom operators, see section 2.3.3.

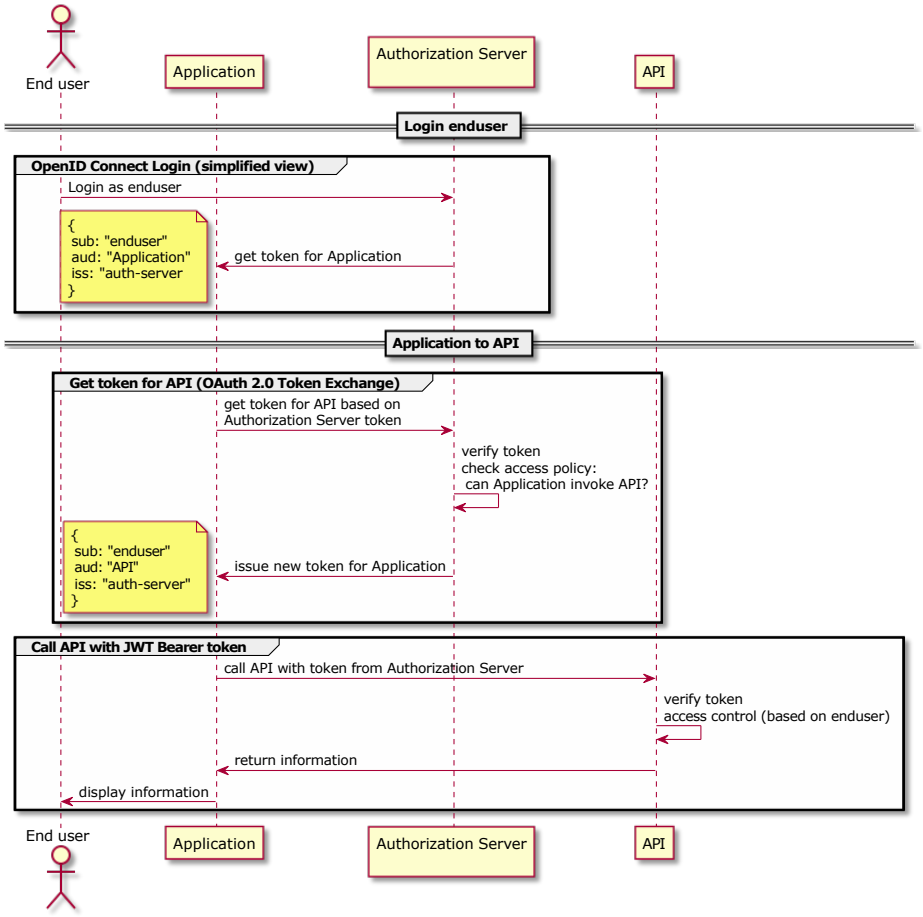

Figure 2.16: On-Behalf-Of Request Flow

Chapter 3

Requirements

In the two previous chapters, different technologies, frameworks and protocols that are relevant to the thesis were discussed. The following will be a description of the requirements that the *Proof of Concept* extension filter were developed from.

3.1 AuthN/Z and Software Development at NAV

> **NOTE**
>
> The authentication protocol OpenID Connect relies on delegated authorization over OAuth 2.0. This can be confusing when discussing authentication vs authorization in the context of an end user. Following, when authentication is discussed delegated authorization is implied as well.

Authentication and authorization are both important cross-cutting concerns, and NAV develops systems that require strict and correct usage of both. The difference though, is that authorization is much more tightly coupled to the business logic. This is reasonable as identity is more or less the same across contexts, but authorization is highly context dependent, access semantics can change a lot between different application domains. The market indicate this as well, authentication services are becoming an industry [8], and in Norway the Norwegian Digitalisation Agency (DigDir) has created common authentication solutions for individuals [154] and services [155]. There are projects and platforms that are working on tooling and standards for authorization and access control [117], but another approach, arguably more suited for cloud native development, is to implement the authorization and access control logic in the application business logic [23], enabling security tests as part of the CI/CD

pipeline. Authorization close to the domain logic is the direction NAV is moving as well, though not without challenges like consistency and access semantics across development team boundaries. Through discussions with NAV security engineers at the start of the thesis work some guiding principles for work on creating new Zero Trust Architecture components were formed, note that these principles are consistent with DevOps values (note 2.3 and 2.4).

- The product teams are responsible for their products security. Other teams like Security Operations Center (SOC) and platform teams are responsible for creating and supporting secure platforms and tooling to build products upon, but the developers in the product teams have the final responsibility for the security of what they create.
- Distribute what can be distributed. Centralized security services create organisational bottlenecks and have high risk of becoming security blackboxes for the product teams, increasing the risk that the team does not understand the security posture of their products. Rather the aim is to enable the product teams to take responsibility, in way that limits risk and increases efficiency as much as possible.
- The platform teams, see 2.4, create *as-a-Service* platforms and tooling that the developers in product teams develop upon. Platform and tools development is in alignment with the goal of a *minimum-viable-platform* [143]. The security model, and the security goal of the platform is Zero Trust Architecture.

3.2 Zero Trust Architecture for Cloud Native Organisations

NAV has since moving to cloud native development implemented a layered approach to Zero Trust Architecture. On the platform level the organisation uses service mesh supported service to service mTLS to secure and provide service level access control. Additionally on top of the service mesh NAV runs a token-based Zero Trust Architecture (see 2.11.3). The design of the token-based approach is that the platform implements and manages authorization servers for token exchange, integration with external and internal identity providers and client registration. The product teams are responsible for implementing clients that conform to the protocols. The platform team supplies documentation and shared libraries to make it easier for developers to implement secure clients. NAV's interest in this thesis is insight into if some parts of the client-side of token-based Zero Trust Architecture can successfully be implemented in the service mesh. This would ensure better consistency and allow the platform team to move faster with security development across the hundreds of services and applications NAV develops and manages. The thesis will focus mostly on implementing authentication as it is the fundamental underpinning for Zero Trust Architecture and is the first thing that needs to be designed and tested

for service mesh implementation. The situation NAV finds itself in is similar to many other organisations that are maturing in the cloud native ecosystem. Insight into how Zero Trust Architecture can be built using security tokens and service mesh would be interesting for many organisations.

3.3 Authentication in-application vs in-mesh

One viable approach is to implement authentication in the application source code. This approach makes the application developer aware and in control of the authentication, but as discussed in section 1 and 2.10 there are security and efficiency issues with the approach. Figure 3.1 and 3.2 presents a view of the two options on where to place the main responsibility of authentication.

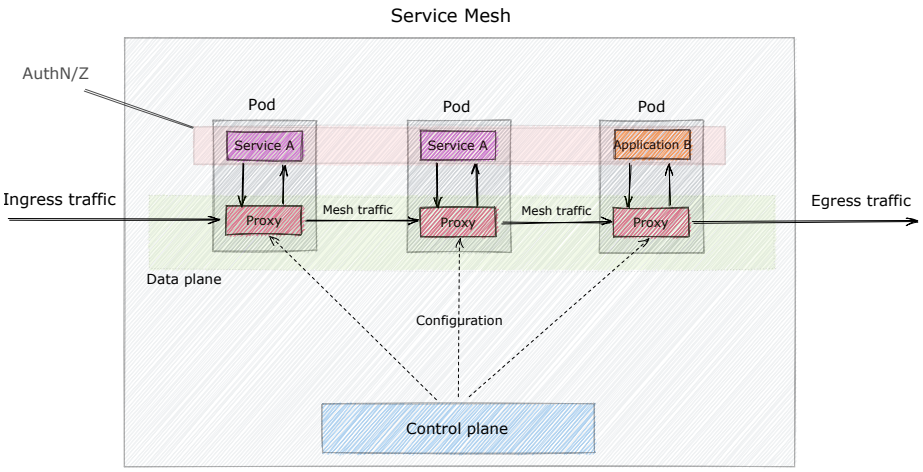

Figure 3.1: Simplified diagram of service mesh with AuthN/Z in implemented in service code

3.3.1 Authentication in application

Figure 3.1 illustrates authentication implemented directly in the services and applications running in the service mesh. In this case every product team needs to implement the OpenID Connect and OAuth 2.0 specifications correctly and understand the implications of the different flows and extensions. Research into vulnerabilities and flaws shows that web application software often has flawed authentication implementation. In the latest version of the OWASP top 10, a well-renowned community lead effort on application security, broken authentication ranks as the second highest risk developers face [121]. On the plus side, the approach necessitates that the developers become more aware of the security of the system, raising the security awareness of the whole team.

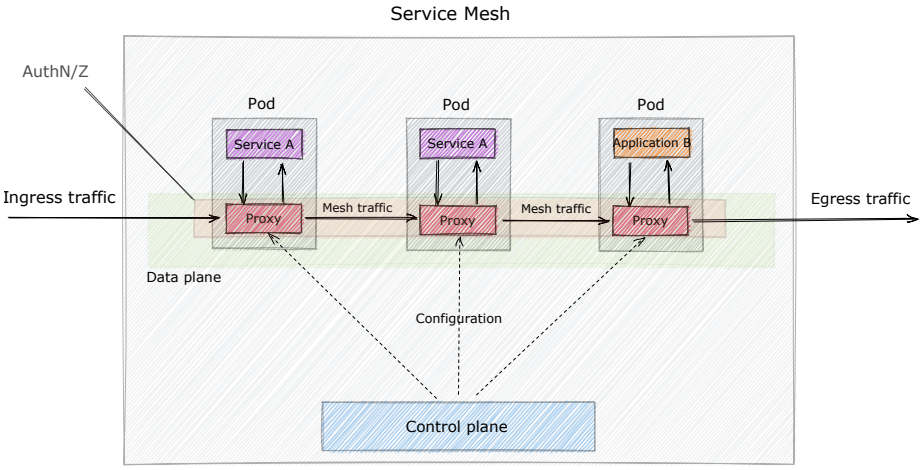

Figure 3.2: Simplified diagram of service mesh with AuthN/Z in implemented in mesh proxies

3.3.2 Authentication in the mesh

Figure 3.2 illustrates authentication implemented in the mesh. Authorization in this case means the end user delegated authorization like in the case of user information in OpenID Connect, see 2.10.

3.4 Authentication in the Service Mesh

The service mesh provides an opportunity for executing the authentication process close to the individual services. Execution is distributed, while implementation is centralized. Avoiding creating another centralised security service, at the same time the code that implements the authentication flow only needs to be implemented, tested and integrated once. Leveraging the service mesh for authentication makes it possible to distribute execution, while centralising risk of implementation errors. Implementing authentication close to the services can be accomplished in different ways with different trade-offs. Through the background research done for the thesis three different approaches were discovered and it is these approaches that we will examine, explore and analyse for usage in NAV's service mesh. The results and analysis should be generalizable to other large organisation's platforms.

Upstream Headers The usual pattern when using some sort of proxy authentication in front of a service or application is that the proxy performing the authentication forwards the security tokens by appending them to the request headers when the request passes through the proxy [112]. Appending security tokens to the request headers allows for upstream service validation of the to-

kens, and the service can perform access control. Typically the proxy embeds the security token in the `Authorization` header [101], or `X-Forwarded-<token type>` headers [102].

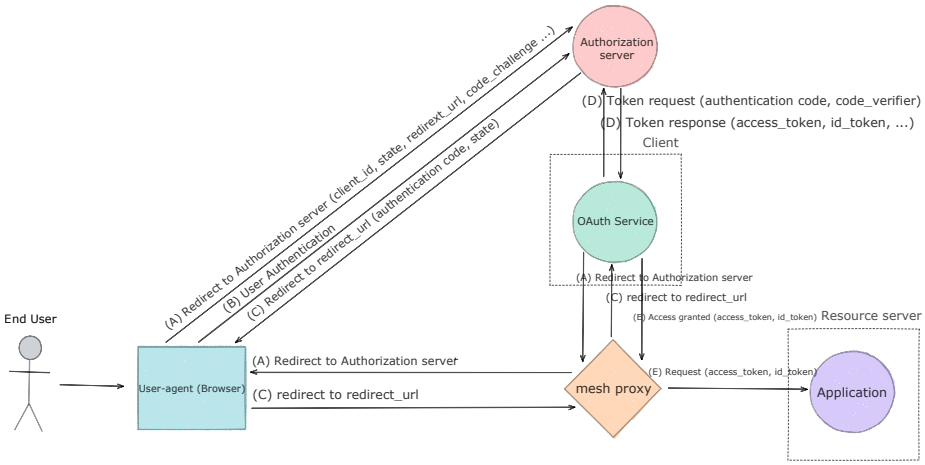

Figure 3.3: OAuth 2.0 Authorization flow with proxy using external authentication

3.4.1 Authentication using External Authorization

Figure 3.3 presents an example of OAuth 2.0 flow using extended authorization. Service mesh implementations, by way of the data plane proxy, commonly offer interfaces for integrating authorization and/or authentication service into the proxy filter chain. Example of this is Envoys built-in filter *External Authorization* [28], used by production ready services like Istio's *authservice* [52]. Analysing the performance, operational and security aspects gives a view of the pros and cons with the approach.

Pros

- **API** The data plane proxies provide the integration point with a well-defined API for authentication services to connect to. Allowing for mixing and matching solutions, or creating custom solutions, that interplay well with the data plane proxy.
- **Service mesh native** This approach has established open source solutions ready to be integrated in the service mesh. Ex. Istio's *authservice* [52].

Cons

- **Latency**, Every request that is serviced by the filter incurs at least one extra network hop, more if caching is used.
- **Resiliency** Additional service that needs to be orchestrated alongside the data plane is a potential point of failure.
- **Security** The authentication service is a security critical service that needs to be monitored, updated, and if compromised due to vulnerabilities or supply-chain attacks pose a serious risk to the organisation. Using an extra service for authentication increases the attack surface of the whole system. Additionally the communication between the data plane proxy and the service must be secured.

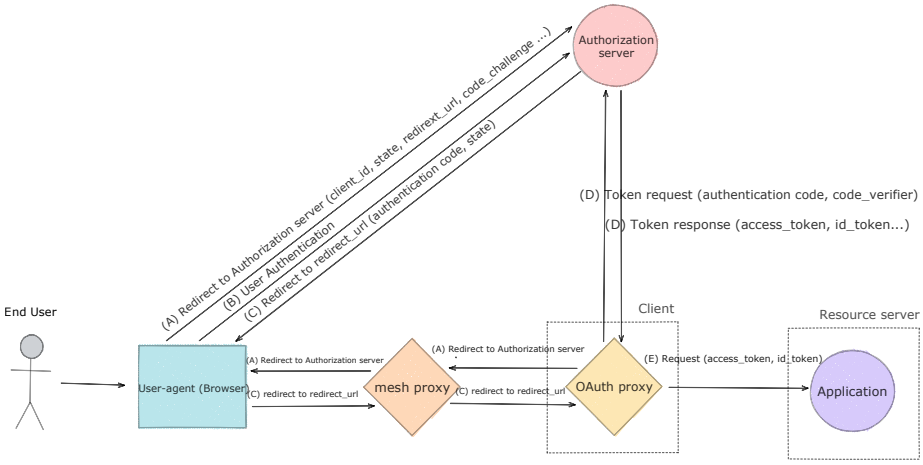

Figure 3.4: OAuth 2.0 Authorization flow with extra oauth proxy

3.4.2 Authentication using Extra Proxy

Figure 3.4 illustrates authentication implemented in an auxiliary process proxying between the service or application and the service mesh proxy. The extra proxy, from here on referred to as *the OAuth proxy*, provides OAuth 2.0 and OpenID connect features. The upstream service receives only authenticated requests. Analysing the performance, operational and security aspects gives a view of the pros and cons with the approach.

Pros

- **Low barrier to entry** This is probably the easiest starting point approach to get authentication support into the service mesh. A *minimal-viable-product* could be to just document for product teams how to set up the solution and iterate from there.

- **Mature** There exists professional, battle tested, solutions for these kinds of proxies like *OAuth2-Proxy* [111] that are well documented for many use cases.

Cons

- **Integration with the service mesh** The service mesh provides no support for configuration and most importantly, dynamic runtime updates to the configuration of the OAuth proxy. Configuration like certificates, trust stores or OpenID Connect configuration. Platform teams that wish to make the OAuth proxy approach a seamless platform component need to develop their own tooling and processes. This might involve some custom software as configuration values should still be the responsibility of the product teams, going back to the principles stated before.
- **Latency** The OAuth proxy approach requires extra network hops between the mesh proxy and the service or application upstream. The latency penalty of the OAuth proxy approach will only be worse with use of session cache, which is usually used when authenticating for security conscious applications.
- **Security** The OAuth proxy is a security critical component that needs to be monitored, updated, and if compromised due to vulnerabilities or supply-chain attacks pose a serious risk to the organisation. Using an extra proxy for authentication increases the attack surface of the whole system. Additionally the communication between the data plane proxy and the OAuth proxy must be secured.

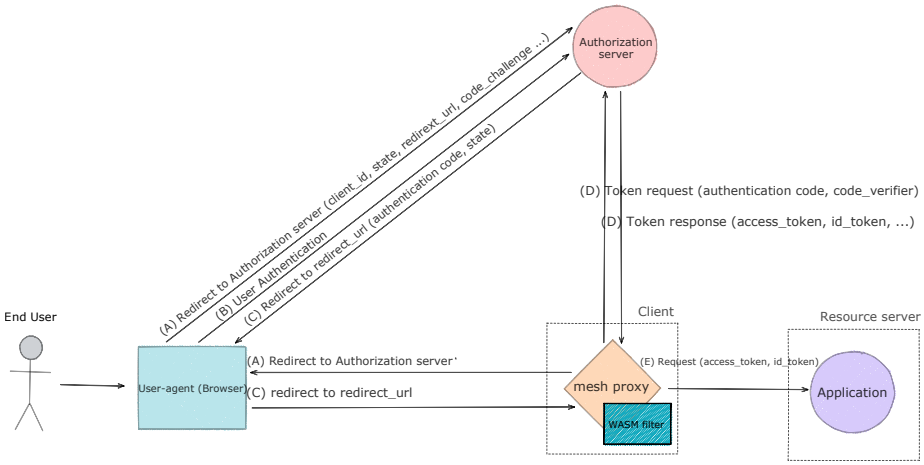

Figure 3.5: OAuth 2.0 Authorization flow with WebAssembly Extension Filter

3.4.3 Authentication using WebAssembly Extension filter

Figure 3.5 show the authentication using an authentication filter module running embedded in the data plane proxy. A benefit, obvious from the visual representation, is the reduction in moving parts. Analysing the performance, operational and security aspects gives a view of the pros and cons with the approach.

Pros

- **In-process integration** The authentication control is contained to the data plane proxy, except for the authorization server, or a session cache, no other process needs to be called over the network for the filter to fulfill its role.
- **Service mesh managed** Service meshes rely on having robust and secure communication between the control plane and the data plane. The filter, being part of the data plane proxy, is configured and kept in sync with updates through the service mesh control plane.
- **Security** The extension filter is delivered as a WebAssembly module, and it enjoys the security benefits of WebAssembly (see 2.8.3). The filter through WebAssembly's security model has stronger mitigations against supply-chain attacks, code execution attacks and other elevation of privilege attacks [135] through the strong sandboxing.

Cons

- **Performance** Extending the proxy means more work for the data plane proxy process, potentially impacting performance
- **Maturity** WebAssembly extension filters are bleeding-edge (October 2020, first proxy support [56]). The ecosystem has not had a long time to develop.

3.5 Proof of Concept Filter

Looking at the different options for listed in section 3.4 extra network hops are necessary for both the external authorization and the extra proxy approach. No matter the innovation and development in the space that will be a constant latency and reliability concern. Additionally being extra processes that needs to be configured an secured is costly for a large service mesh of services. The authorization service or proxy could be centralized to lower the operational cost, but then the architecture runs across the principles defined in the start of this chapter. Authentication using extension filters, while bleeding-edge technology, has a great potential. Configuration can be managed without ceremony through the service mesh data plane and WebAssembly provides relatively unique security and reliability properties that are highly valued in an authentication component.

Proof of Concept risk

- Is the WebAssembly ecosystem mature enough to create a suitably robust and reliable filter implementation?
- Is the *proxy-wasm ABI* (see 2.7.4) specification mature enough for production usage?

3.6 Proof of Concept Filter Requirements

The filter needs to support the OAuth 2.0 and OpenID Connect features needed for a *minimal-variable-product*. Together with security engineers at NAV the following requirements for the Proof of Concept filter were specified. The requirements are divided into two categories, *functional* and *non-functional*.

3.6.1 Functional requirements

R1 **OAuth 2.0 and OpenID Connect** A service mesh solution needs to support OAuth 2.0, with OpenID Connect for authentication. As described in section 2.9 and 2.10 OpenID Connect extends OAuth 2.0 with authentication capabilities. OpenID Connect has become the most used authentication protocol and is the protocol used by NAV and the rest of the Norwegian public sector at large through *ID porten* [154]. The PoC filter needs to support the specification of OAuth 2.0 clients (RFC 6749) [16] and OpenID clients [133].

R2 **Authorization Code Flow with PKCE** The NAV platform security team requires that the OAuth 2.0 is only used with Authorization Code flow, see 2.9, and with the PKCE (RFC7638) extension [105]. PKCE extension mitigates against access code interception or leakage.

R3 **OpenID Connect Token Validation** OpenID Connect client guide [133] states that the client is required to validate the ID Token received from the OpenID Provider. In the separation of duty between the proxy and the upstream service, the proxy is responsible for the validation before forwarding the token to the upstream service.

R4 **Configurable through the service mesh** The OAuth 2.0 and OpenID Connect configuration options and additional filter configuration options should be configurable through service mesh configuration.

3.6.2 Non-Functional requirements

R5 **Performance** The Proof of Concept filter should be comparable in performance tot the alternative approaches. Preferably better as the filter is communicating with the proxy over process boundaries, not network connection.

R6 **DevOps** The Proof of Concept filter needs to be easy to configure through mechanisms used to configure and manage the service mesh. Meaning it should not require additional configuration or management processes. The Configuration should be intuitive for a platform engineer familiar with the OAuth 2.0 and OpenID Connect domain.

R7 **Security** The filter implementation and integration with the proxy should not incur any new noteworthy security risks in comparison to implementing the protocols directly in application code.

3.7 Summary

The summary of the following chapter is that a *Proof of Concept filter* will be implemented supporting a subset of a full token-based Zero Trust Architecture. The subset to be implemented is authentication and delegated authorization in the service mesh data plane. the *Proof of Concept filter* needs to provide a secure and performant implementation of the OAuth 2.0 and OpenID connect flows, be easy to configure for the specific environment it is to be used in, and be easier for developers to work with than implementing the flows themselves.

Chapter 4

Design

> Life is pain, Highness! Anyone who says differently is selling something.
>
> *Man in black, The Princess Bride*

The requirements guide the design of the Proof of Concept Filter (PoC filter). The Poc filter is an WebAssembly module to be run as an extension filter for a *proxy-wasm ABI* host. The PoC filter provides OAuth and OpenID Connect services. The module approach should provide advantages in the following areas:

- **Performance** The PoC filter is directly connected to the data plane proxy through the WebAssembly host, providing near native performance.
- **Security** The module implements the authentication and authorization flows once for the whole service mesh. WebAssembly provides security features that are good for extensions, like capability based security, sandboxing, isolation and no undefined behaviour, see 2.8.
- **DevOps** Leveraging the already in use service mesh proxies, with service mesh support for configuration.

4.1 Architecture Overview

As described in section 2.7.4 the proxy that handles the requests in a service mesh data plane is composed of several internal components that work together to proxy the request correctly. Figure 4.1 shows an example where Envoy is used, but the important piece is the *proxy-wasm ABI* specification that connects the proxy to the filter module. The *proxy-wasm ABI* creates a standardized interface meaning the filter module would work for any proxy that supports the interface. Looking at figure 4.1 the *listener* is responsible for handling incoming

requests and creating the *filter chain* based on configuration. If the configuration specifies WebAssembly filters, the proxy is responsible for instantiating and managing the WebAssembly virtual machines used by the filters. As illustrated in the figure the module implements handler functions (exports in WebAssembly language) and calls functions the proxy exposes (imports in WebAssembly language). The WebAssembly filter module communicates back to the *filter chain* through the return values from the handler functions (see 2.7.4). The PoC filter is represented as the green box in figure 4.1. The design that follows in this chapter describes how the PoC filter implements OAuth 2.0 and OpenID Connect, and how it interacts with the proxy and external services to accomplish that.

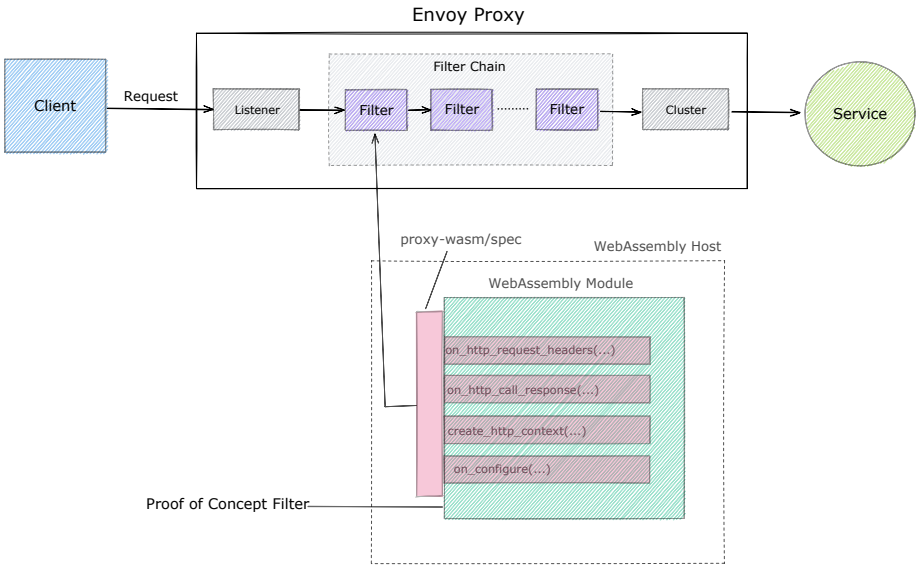

Figure 4.1: Architecture diagram of the Envoy proxy filter chain with WebAssembly module

4.2 Configuration

To make the filter useful it needs to be configurable by the users. OAuth 2.0 and OpenID Connect both define several standard configuration options, such as authorization server / identity provider URI, client id and secret, etc. In addition several of the authorization server / identity providers have implemented or extended the standards in different ways meaning client implementations like the thesis filter needs to allow configuration of options outside the standard. In addition some configuration options are required to use the Envoy proxy for HTTP requests from the filter directly.

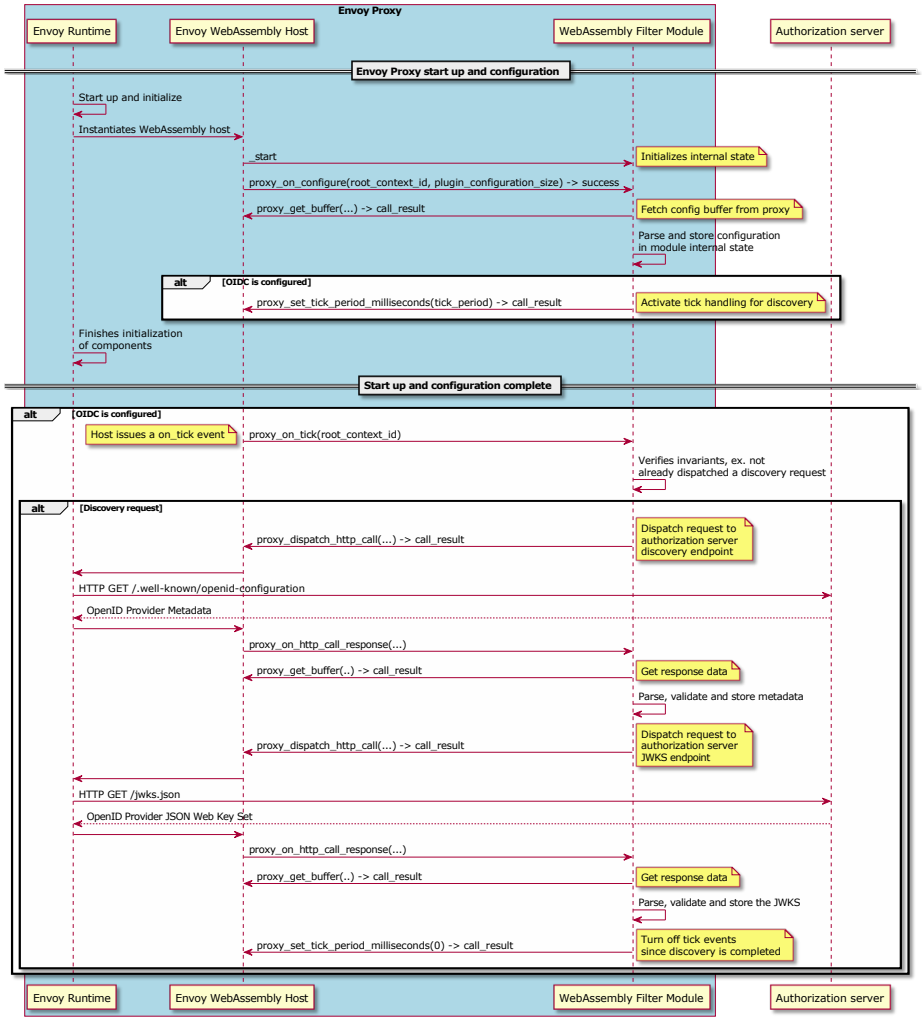

Figure 4.2: WebAssembly filter module start up and configuration sequence diagram

4.2.1 Startup and Configuration

If the filter module is configured for OpenID Connect additionally configuration steps need to be performed after the Envoy Proxy is done initializing and configuring all components, primarily the filter module requires the authorization server cluster to be up and running, see section 2.7 for description on how clusters work in Envoy. The filter module requires that the authorization server implements OpenID Connect Discovery [104]. The following describes how the filter module is initialized and configured, see figure 4.2 for visualization of the sequence.

1. The Envoy proxy starts up, parses configuration and initializes different internal components including filters.

2. When a WebAssembly filter is configured, Envoy creates a WebAssembly host for each worker thread running in Envoy internally, see section 2.7.4, and loads the configured module.

3. The module(s) *_start* function is called by the Envoy WebAssembly host. The *_start* function is used by the modules to initialize state.

4. The *proxy_on_configure(root_context_id, plugin_configuration_size)* is called when the configuration for the filter is ready to be fetched. And the filter module calls *proxy_get_buffer(...)* to retrieve a buffer with the raw configuration data. The configuration data is parsed and validated. If the data is malformed the module returns a negative result back to the proxy indicating that the configuration failed.

5. If the filter is configured for OpenID Connect (The scope configuration contains a openid) the module activates the *on_tick(context_id)* event by calling the *proxy_set_tick_period_milliseconds(tick_period)* function import with a positive, non-zero value.

4.3 Authorization Flow

The OAuth 2.0 working group recommends implementers of OAuth 2.0 clients and authorization servers to only use Authorization Code flow for end-user facing clients [84]. The extensions developed in this thesis will support OAuth 2.0 Authorization Code flow with PKCE as recommended by the OAuth 2.0 working group, and OpenID Connect Authorization Code flow with PKCE.

The filter is designed with inspiration from *OAuth2 Proxy* [111] and Istio's Authservice [52]. To handle the different stages in the Authorization flow the filter implements endpoints in the form of HTTP paths that handle a stage in the flow. This is similar to how an application would do it, but ex. Authservice uses a state based design to complete the authorization flow. The reason this design was not chosen for the thesis filter is that the path-based design is simpler to implement and test as will be shown in section 5. The potential drawback

is that the filter "hijacks" HTTP paths that applications and services upstream might use. This means that developers need to be aware of what paths the filter takes ownership of.

4.3.1 Unauthenticated/Unauthorized Requests

After the filter has started up and initialized the required state it is ready to receive requests. Figure 4.3 shows the flow of events inside the Envoy proxy and between the end user and authorization server. The sequence diagram shows the lifetime of the filter chain in the Envoy runtime and the lifetime of the HTTP Context in the filter as active bars. Note that the OAuth authorization code flow shown in the diagram involves two independent HTTP Contexts.

1. The process starts with an end user that issues a HTTP request to the proxy standing in-front of an upstream service or application. Envoy starts with creating a filter chain that in this case includes the WebAssembly filter module.

2. The filter module gets issued a *proxy_on_context_create(context_id, parent_context_id)* call to create a new HTTP Context that filters the request.

3. Later when the Request headers are received Envoy the *proxy_on_http_request_headers(...)* event handler is called in the module ending up in a call to the HTTP Context created previously. The HTTP Context fetches the headers data from Envoy and checks for active sessions, since the request is Unauthenticated it lacks an active session.

4. The HTTP Context creates a new session containing an identifier and OAuth, and if applicable OIDC state. Finally the HTTP Context calls *proxy_send_http_response(...) -> call_result* with a request to redirect the end user to the authorization server, starting the Authorization Code Flow.

5. The end user is redirected to the authorization server and performs the authorisation/authentication request.

6. After a successful authorization/authentication request with the authorization server the end user is redirected back to the Envoy proxy. This triggers the creation of a new filter chain which creates a new fresh HTTP Context.

7. The process is the same as 3 but it this case there is a session in the cache which is used to match and validate the parameters passed with the redirect from the authorization server. If the validation fails the filter returns an error response back to the end user using *proxy_send_http_response(...) -> call_result*.

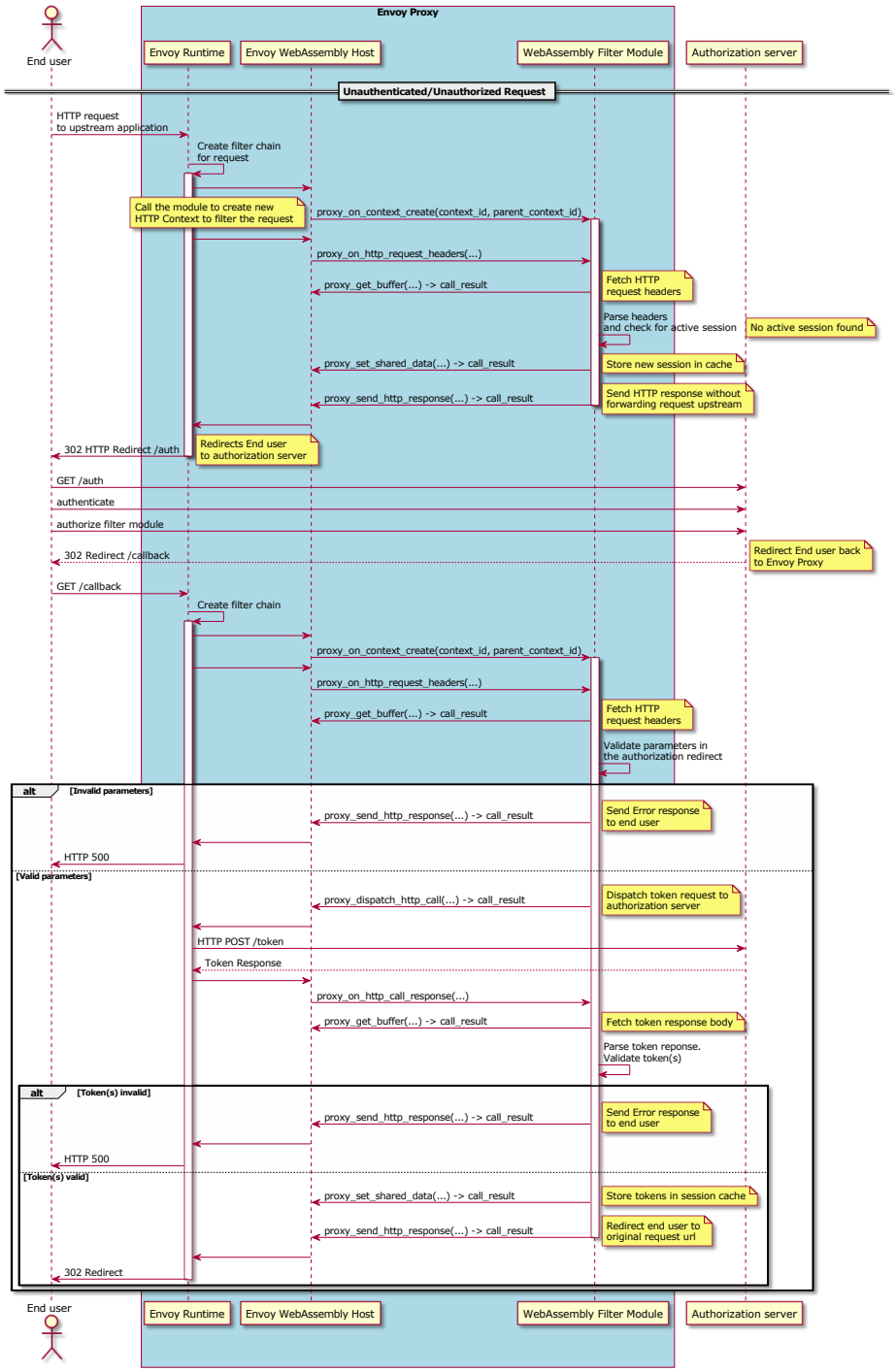

Figure 4.3: Sequence of events and actions in filter module handling an unauthenticated/unauthorized request

8. If the callback is HTTP, Context issues a token request to the authorization server using *proxy_ dispatch_ http_ call(...) -> call_ result*.

9. When the token response arrives, in the case of an id token is validated according to the OpenID Connect Core specification, See section 2.10. If the token(s) or any other of the fields in the token response is invalid an error response is sent to the end user.

10. If the token(s) are valid the session cache is updated with the tokens to be used for subsequent calls during the active session. Lastly the end user is redirected to the original request URL from the start of the authorization code flow.

4.3.2 Proxied Requests

Sessions with end users that have been through the Authorization Code flow and have valid tokens are proxied to the requested upstream resource. Figure 4.4 illustrates how the request is handled by Envoy and the filter module.

1. The end user issues a HTTP request to Envoy, meant for an upstream application. Envoy instantiates a filter chain, requesting a new HTTP context from the filter module.

2. When the HTTP headers arrive *proxy_ on_ http_ request_ headers(...)* is called on the HTTP Context in the filter module. The HTTP Context fetches the headers data.

3. The request headers are parsed and checked for active sessions. As this is an authenticated and authorized session there are valid security tokens in the session.

4. The HTTP Context appends the tokens to the request headers and returns a *Continue* action to Envoy indicating that the filter chain can continue.

5. Envoy finishes processing the filter chain. If all filters return *Continue* actions the request is clear to continue upstream. Envoy proxies the request to the upstream application.

6. The upstream application receives the request, does the processing required and returns a response. The response is handled by Envoy. Not shown in the diagram is that the filters that handled the request can also handle the response. Note that the HTTP Context is still active until the response is returned to the end user and the cycle is complete.

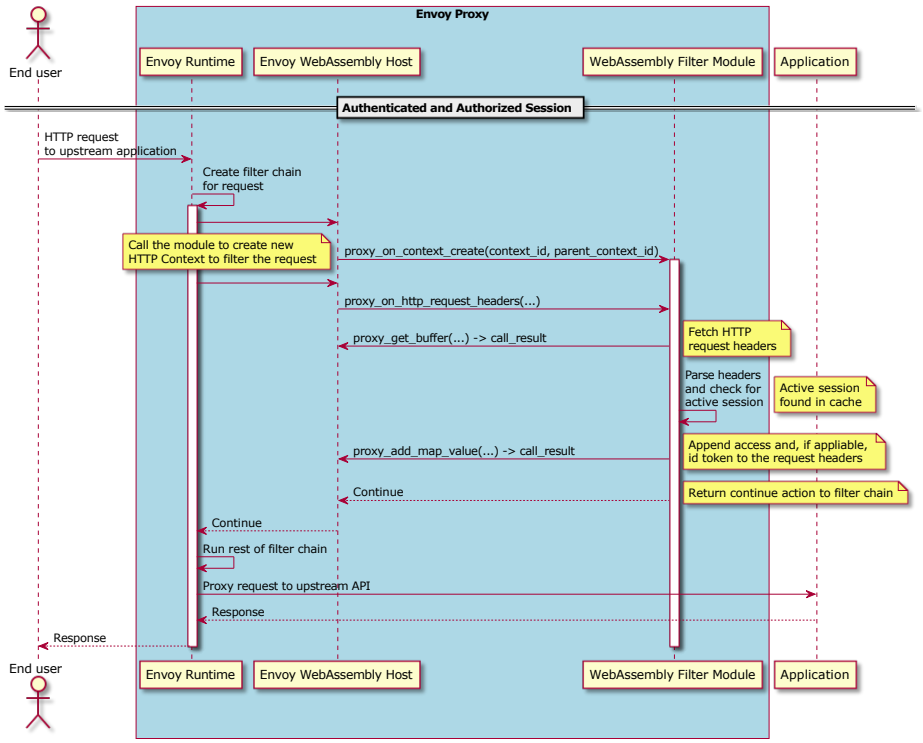

Figure 4.4: Sequence of events and actions in filter module handling a proxied request

4.4 Upstream

Upstream application or service needs to receive access and id tokens from the proxy to be able to do additional domain specific authorization or other end-user specific logic. To support this the extension will forward the successfully minted access token and id token upstream through headers. In addition to the standard HTTP `Authorization` header [101]. The filter appends, depending on configuration and received tokens, `X-Forwarded-<token type>` headers [102].

- **Authorization** contains the access token received from the authorization server for the current request. The Authorization header will have the *Bearer* type.
- **X-Forwarded-Access-Token** contains the access token received from the authorization server for the current request.
- **X-Forwarded-ID-Token** contains the id token verifying the identity of the end-user of the request.

4.5 Summary

In this chapter the design of the *Proof of Concept filter* has been described. The filter needs to handle *proxy-wasn ABI* events like startup and receiving configuration data. The core part of the design is how the OAuth 2.0 and OpenID Connect protocols will be implemented using the ABIs, provided by the specification, to interact with the data plane proxy. Finally filter will forward the security tokens upstream through standard HTTP headers allowing upstream services and applications to also verify tokens, and potentially perform access control, or call additional services.

Field	Type	Default	Description
redirect_uri	String	/callback	URL the authorization server redirects the end-user back to after authentication.
cookie_name	String	oidcSession	Cookie name that holds the session cookie for the end-user session with the filter.
client_id	String	**Required**	OAuth 2.0 / OIDC client ID.
client_secret	String	**Required**	OAuth 2.0 / OIDC client secret.
issuer	String	**Required**	Authorization Server identifier. Required for token validation when using OpenID Connect.
auth_uri	String	**Required**	The URL that the filter will issue token requests against.
auth_cluster	String	auth_server_cluster	Envoy cluster that the filter will use to issue token request to the authorization server / identity provider
extra_params	list[[String, String]]	[]	Extra query parameters the filter will add to the authorization redirect to the authorization server. Necessary with some authorization server / identity providers
scopes	String	openid	OAuth 2.0 scopes.
cookie_expire	Integer	**Required**	Expiry time for the end-user session

Table 4.1: Filter Configuration Options

Path	Description
/start	Request to this path triggers starting a new OAuth 2.0/OIDC authorization flow.
/callback	path on the proxy the authorization server redirects the end-user back to after authentication. The filter will treat all requests to this endpoint as authorization callback redirects. The filter will expect certain parameters are present in the request, and respond with an error response if the redirect is invalid.
/sign_out	Clears the session with the filter, does not clear the session with the authorization server/identity provider

Table 4.2: Filter Endpoints

Chapter 5

Implementation

5.1 Tools, Languages and Libraries

5.1.1 Scope Limitation

In this thesis we will primarily be looking at the Istio service mesh and the Envoy proxy due to time concern. Istio and Envoy are leading the market in adoption at the time of writing [149]. Envoy is also further ahead in implementing the WebAssembly extension standard for proxies, but Linkerd has the feature on its road map so feature parity seems to be coming [39]. The thesis will look at other implementations to compare when suitable for the analysis. Note that this technology area is under heavy development and competition so we will focus not on the specific implementations, but rather the standards and concepts that seem to be gaining ground and solving problems for users.

5.1.2 Programming Language

Rust is a systems language that compiles to native binaries. It is fast, has low runtime overhead, both in terms of size, speed and complexity, uses little space and have a small memory footprint at runtime. Additionally the language is designed to be memory safe, avoid accidental mutation of state, null referencing and data races. Rust is an excellent language for systems programming in general but it also has "best-in-class" support for WebAssembly programming. Taken together it made the natural choice to develop the *PoC filter*.

Rust Core Values [131]

- **Performance** Blazingly fast, memory efficient, no runtime or garbage collector. Can power performance-critical services, run on embedded devices and can easily integrate with other programming languages.
- **Reliability** Rust's rich type system and ownership model guarantee memory-safety and thread-safety, eliminating many classes of bugs at compile-time.

- **Productivity** Great documentation, user friendly compiler with useful error messages, and top notch tooling. Includes an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspection, auto-formatter etc.

5.1.3 WASI

Envoy supports running plain WebAssembly modules and WASI modules, see section 2.8 more on WASI. The thesis filter was originally compiled to plain WebAssembly, compile target `wasm32-unknown-unknown`, but as the implementation progressed the need for generating secure random numbers (CSPRNG) [73] for fields like state and PKCE in the OAuth 2.0 specification. The filter had to switch compilation target to `wasm32-wasi` to get support for host call [9]:

```
1 fn random_get(buf: *mut u8, buf_len: Size) -> Result<(), Errno>
```

Which enables creation of CSPRNG numbers by way of WASI supported WebAssembly hosts.

5.1.4 Dependencies

The PoC filter uses software library dependencies that are included at compile time. In Rust a library or executable program is called a crates, crates are compiled using the Rust compiler, *rustc* [156]. The complete *Cargo.toml*, which defines dependencies and compilation configuration is available in Appendix A.2.

proxy-wasm The PoC filter uses the `proxy-wasm` crate [138] to interface with the Envoy WebAssembly host environment. The library is developed and maintained by the same group behind the Envoy implementation of WebAssembly extensions and the `proxy-wasm ABI`. The library provides an abstraction layer over the raw ABI. The SDK exposes Traits with extendable methods to create filters for Envoy. Core Traits are; `Context`, `HttpContext`, `RootContext`, and `StreamContext` which lets developers create filters that handle the different classes of *proxy-wasm ABI* events. The SDK implements no-op stubs for the functions a module does not implement explicitly. Additionally the SDKs expose a more ergonomic API for developers to implement modules.

oauth2 The crate `oauth2` [14] is an an extensible, strongly-typed implementation of OAuth2 (RFC 6749) [16]. The PoC filter uses the crate to craft OAuth and OpenID redirect urls with correct structure and content, including state, code challenges etc. Additionally the crate exposes functions and Traits for crafting token requests and validating OAuth 2.0 invariants.

jwt-simple The crate `jwt-simple` [17] is A new JWT (JSON Web Tokens) implementation for Rust that focuses on simplicity, while avoiding common JWT security pitfalls. The crate uses only pure Rust implementations, and can be compiled out of the box to WebAssembly/WASI. The WebAssembly support is important as many of the common Rust JWT crates have dependencies that don't support WebAssembly + WASI, see 5.2.4. The PoC filter uses the create for its convenient API for deserializing JWT strings to Structs, and for validation of JWT signature.

jsonwebkey The crate `jsonwebkey` [50] is JSON Web Key (JWK) (de)serialization, generation, and conversion crate. The crate is used by the filter to deserialize JWK received during the OpenID discovery flow. Used together with `jwt-simple` to validate received ID tokens from the authorization server.

5.2 Proof of Concept OAuth Filter

5.2.1 Project Modules

See Appendix A.1 for the PoC filter source code. The complete source code and unit tests are located in the *src* directory. The modules are structured and named after module responsibility and function.

src/

`cache.rs`
Session cache logic, serialization and deserialization

`config.rs`
Handling of raw configuration passed from Envoy and representation of internal configuration.

`discovery.rs`
OpenID Connect Discovery, helper functions for crafting discovery requests and parsing responses.

`filter.rs`
Main entry point into the filter. Responsible for integrating the OAuth and OpenID Connect domain code with the Envoy *Proxy-Wasm ABI*.

`messages.rs`
Structs encapsulating domain messages to authorization server, end user and upstream.

`mock_overrides.rs`
Stub implementations of *Proxy-Wasm ABI* so unit tests can be compiled and executed in `x86_64`. See 5.2.3.

`oauth_client.rs`
Main domain module, implements the OAuth and OpenID Connect flows. Called by `filter.rs`.

`oauth_client_types.rs`
Structs and Traits representing domain concepts in OAuth and OpenID Connect, used by `filter.rs` and `oauth_client.rs`.

`session.rs`
End User session handling Structs and functions. Features session cookie crafting, assertion logic and serialization and deserialization of session data to and from the cache.

`util.rs`
Auxiliary code that is reused multiple places in the code base.

5.2.2 Filter Core Components

OAuthRootContext

Struct `OAuthRootContext` responsibility is to make sure the filter is correctly configured, and creating new `OAuthFilter` instantiations, `OAuthFilter` handle the received request. As described in section 4 the filter can be configured for plain OAuth or OAuth with OpenID Connect. The `provider_metadata`, `jwks` and `request_active` fields are only used when the filter is configured for OpenID Connect.

```
1 struct OAuthRootContext {
2     config: Option<RawFilterConfig>,
3     provider_metadata: Option<ProviderMetadata>,
4     jwks: Option<JsonWebKeySet>,
5     request_active: bool,
6 }
```

Source Code 5.1: OAuthRootContext Struct

Filter startup As Source Code 5.2 shows, at filter startup the `_start` function is called by the WebAssembly host as defined in *Proxy-Wasm ABI*, see 2.7.5. The `_start` function registers an instance of the Struct `OAuthRootContext` as the root context of the filter. The correct matching of event calls from Envoy to the correct context is handled by the SDK, see 5.1.4. Note that `OAuthRootContext` is instantiated with `None` as the value of the config field, configuration data is received in the `on_configuration` event which is called later in Envoys initialization process.

```
1 #[cfg(not(test))]
2 #[no_mangle]
```

```
3  pub fn _start() {
4      proxy_wasm::set_log_level(LogLevel::Debug);
5      proxy_wasm::set_root_context(|_| -> Box<dyn RootContext> {
6          Box::new(OAuthRootContext {
7              config: None,
8              provider_metadata: None,
9              jwks: None,
10             request_active: false,
11         })
12     });
13 }
```

Source Code 5.2: Initialization function called at filter startup

OAuthFilter

An `OAuthFilter` instance is created to manage each individual request lifecycle. The Struct mainly serves as a mediator between Envoy events and `OAuthClient`. It is also responsible for translating OAuthClient responses into *proxy-wasm ABI* calls. Source code 5.3 shows the Struct. `config` is validated configuration data passed from `OAuthRootContext` on creation.

```
1 struct OAuthFilter {
2     config: FilterConfig,
3     oauth_client: OAuthClient,
4     cache: RefCell<SharedCache>,
5 }
```

Source Code 5.3: OAuthFilter Struct

Receiving Requests `OAuthFilter` Handles receiving new requests from downstream end users. As described in section 2.7.4 This is accomplished by implementing the handler function exposed by *Proxy-Wasm ABI*. `HttpContext` is a *Trait* exposed by the *proxy-wasm SDK*. To handle event functions in the `HttpContext` Trait needs to be implemented. Implementing `on_http_request_headers` means the filter handles when the request headers are available from the request. As line 6 in Source code 5.4 shows to actual get the headers data the filter needs to call the proxy through the SDK exposed function `get_http_request_headers()`. On line 7 the end user session is fetched from the cache based on the request header data, if the headers don't contain session identifiers the `user_session` will be `None`. On line 17 `OAuthFilter::endpoint(&self, request: Request, session: Option<Session>) -> Result<FilterAction, ClientError>` is called. The function passes the request to the correct endpoint handler based on the request *path*. `endpoint` returns a `FilterAction` instance which describes what action the filter should take based on the request.

- `TokenRequest` The filter needs to issue a token request to the authorization server.

- `Redirect` The filter needs to redirect the end user to the contained URL.
- `Response` The filter needs to directly respond to the end user with the contained message.
- `Allow` The end user session is valid, the request should be proxied upstream.

Note that in every case the filter ends with returning a `Action` back to the filter chain.

```
// Implement http functions related to this request.
// This is the core of the filter request handling.
impl HttpContext for OAuthFilter {
    // This callback will be invoked when request headers arrive
    fn on_http_request_headers(&mut self, _: usize) -> Action {
        let headers = self.get_http_request_headers();
        let user_session = self.session(&headers);

        let request = Request::new(headers);
        let request = if let Err(error) = request {
            self.send_error_response(error.response());
            return Action::Pause;
        } else {
            request.unwrap()
        };

        match self.endpoint(request, user_session) {
            Ok(filter_action) => match filter_action {
                FilterAction::TokenRequest(request) => {
                    match self.dispatch_http_call(
                        self.config.auth_cluster(),
                        request.headers(),
                        Some(request.body()),
                        vec![],
                        Duration::from_secs(20),
                    ) {
                        Ok(_) => {}
                        Err(error) => {
                            log::error!(
                                "Failed to dispatch token request to cluster = {}\
                                 Envoy status = {:?}",
                                self.config.auth_cluster(),
                                error)
                        }
                    }
                    Action::Pause
                }
                FilterAction::Redirect(redirect) => {
                    self.respond_with_redirect(redirect.url().clone(), redirect.headers().clone());
                    Action::Pause
                }
                FilterAction::Response(response) => {
                    self.send_error_response(response);
                    Action::Pause
```

```
                }
                FilterAction::Allow(token_headers) => {
                    for header in token_headers {
                        self.add_http_request_header(header.0.as_str(), header.1.as_str());
                    }
                    Action::Continue
                }
            },
            Err(error) => {
                self.send_error_response(error.response());
                Action::Pause
            }
        }
    }
}
```

Source Code 5.4: OAuthFilter implements `on_http_request_headers`

OAuthClient

Implements OAuth2 and OpenID Connect Authorization code flows.

- (RFC 6749) [16]
- (OpenID Connect Core 1.0) [133]

```
pub(crate) struct OAuthClient {
    config: FilterConfig,
    client: BasicClient,
}
```

Source Code 5.5: OAuthClient Struct

Start Start new OAuth / OIDC flow: The client accepts a request from the end-user and returns a redirect URL with the correct parameters. Additionally an update Struct is returned with values to be stored for the session.

```
let (redirect, update) = self.oauth_client.start(request)?;
```

Source Code 5.6: OAuthClient start method

Handle callback Handle a callback from the authorization server: The call accepts the redirect request originating from the authorization server and the end-user session. Returns a token request for the authorization server token endpoint.

```
let token_request = self.oauth_client.callback(request, session)?;
```

Source Code 5.7: OAuthClient callback method

Proxy Proxy Http request for upstream applications: Accepts the end-user session and returns the how the request should be handled based on the session state. `Denied` indicates that the end user should receive a direct 401 Unauthorized response, on `UnAuthenticated` end user should be redirected to the authorization server.

```
match self.oauth_client.proxy(session)? {
    Access::Denied(response) => { .. }
    Access::Allowed(headers) => { .. }
    Access::UnAuthenticated => { .. }
}
```

Source Code 5.8: OAuthClient proxy method

SessionCache

As stated in section 4 *proxy-wasm ABI* exposes APIs for storing and retrieving shared data between running modules in the proxy. The *PoC filter* uses this shared storage for storing user sessions. A natural further development would be to extend the storage capability to connect to external production level caches like Redis [130]. Source code 5.9 shows how the cache is fetched from the proxy host when a new `OAuthFilter` is created to handle a new request. The `SessionCache` is passed to `OAuthFilter` as a constructor parameter. The design has several drawbacks, ex. the all the session data is loaded into the module each time a new HTTP request is received. A future development would be to fetch individual sessions keyed on the active sessions identifier. The field `SHARED_SESSIONS_KEY` is a global constant.

```
pub fn from_host(context: &dyn Context) -> Result<SharedCache, String> {
    let (bytes, size) = context.get_shared_data(SHARED_SESSIONS_KEY);
    if let (Some(bytes), Some(_)) = (bytes, size) {
        let cache: SharedCache = serde_json::from_slice(bytes.as_slice()).unwrap();
        Ok(cache)
    } else {
        Err("No shared session cache created".to_string())
    }
}
```

Source Code 5.9: SharedCache method for fetching session data from the proxy host

The `OAuthFilter` calls the `store` method on the `SharedCache` when new session state has been created or a session should be deleted. Source code 5.10 presents the method that implements the storing. Unfortunately the current method implementation stores the whole state into the proxy making it more and more expensive as more users start sessions. Additionally the implementation has a *race-condition* when two users complete actions that update session state, like when during steps in Authorization Code flow. It was deemed that the implementation was good enough for testing and exploring the *PoC filter*, as mentioned before a new caching mechanism was thought out but it was not prioritised to implement it.

```rust
pub fn store(&mut self, context: &dyn Context) -> Result<(), String> {
    let serialized = serde_json::to_string(self);
    match serialized {
        Ok(serialized) => {
            let result = context.set_shared_data(
                SHARED_SESSIONS_KEY,
                Some(&serialized.as_bytes()),
                None,
            );
            match result {
                Ok(_) => Ok(()),
                Err(status) => Err(format!(
                    "Error from host when attempting to set shared data, status={:?}",
                    status
                )),
            }
        }
        Err(error) => Err(error.to_string()),
    }
}
```

Source Code 5.10: SharedCache method for storing session data in the proxy host

5.2.3 Testing

The filter is compiled to `wasm32-wasi` target to be ran in a WebAssembly host. Unfortunately, at time of writing, testing support in Rust WebAssembly is not mature [10]. Testing with a `x86_64` target like `x86_64-unknown-linux-gnu` or `x86_64-apple-darwin` [157] is much more convenient for CI pipelines and local testing. Unfortunately the *proxy-wasm SDK* cannot, without workarounds, be compiled to other targets because of the external dependencies that the WebAssembly host environment fulfils at runtime. Source code 5.11 is one example of such an external dependency defined in the *proxy-wasm SDK*.

```rust
#[no_mangle]
pub extern "C" fn proxy_on_request_headers(context_id: u32, num_headers: usize) -> Action {
```

```
    DISPATCHER.with(|dispatcher| dispatcher.on_http_request_headers(context_id, num_headers))
}
```

Source Code 5.11: OAuthClient proxy method

For unit tests the filter does not need working imports from the WebAssembly host. Unit tests should test the domain logic implemented in the project, not external functions. So to make testing on convenient platforms possible the `mock_overrides.rs` module stubs out the required imports with noop functions. As Source code 5.12 shows the stub module is only compiled and linked when compiled with the test profile.

```rust
#[cfg(test)]
#[allow(unused)]
pub mod overrides {
    use proxy_wasm::types::{BufferType, MapType, Status};

    #[no_mangle]
    pub extern "C" fn proxy_done() -> Status {
        Status::Ok
    }

    #[no_mangle]
    pub extern "C" fn proxy_http_call(
        upstream_data: *const u8,
        upstream_size: usize,
        headers_data: *const u8,
        headers_size: usize,
        body_data: *const u8,
        body_size: usize,
        trailers_data: *const u8,
        trailers_size: usize,
        timeout: u32,
        return_token: *mut u32,
    ) -> Status {
        Status::Ok
    }

  ... // more stubs
}
```

Source Code 5.12: OAuthClient proxy method

Unit Testing

Unit testing tooling is built-in in the Rust tool *Cargo*. Using the command `cargo test` at the root of the project directory runs all the tests defined in the project.

OAuthClient Unit Test The `OAuthClient` is the most prominent and behaviour rich component in the filter. The methods are tested using fixtures and

assertions. Unit testing allows for incremental development and hinders regression bugs. Rust pattern matching allows for expressive and powerful testing. Source code 5.13 is a unit test of the request handling when the path is not one of the filter owned paths.

```rust
#[test]
fn proxy() {
    let client = test_oauth_client();
    let (request, session) = test_authorized_request();

    // Authenticated and valid sessions are accepted
    let result = client.proxy(Some(session));
    assert!(result.is_ok());
    assert!(matches!(result.unwrap(), Access::Allowed(..)));

    // Empty (first request) session are unauthenticated
    let result = client.proxy(None);
    assert!(result.is_ok());
    assert!(matches!(result.unwrap(), Access::UnAuthenticated));

    // Sessions that are waiting for callback are denied
    let result = client.proxy(Some(test_callback_session().1));
    assert!(result.is_ok());
    assert!(matches!(result.unwrap(), Access::Denied(..)));
}
```

Source Code 5.13: OAuthClient proxy test method

5.2.4 Problems and Challenges

WebAssembly Ecosystem Maturity

The WebAssembly ecosystem of software packages and host environments is new and the technology is under heavy development. As follows the ecosystem is not fully mature, this also impacted this project. As part of implementing support for id token validation as specified in OpenID Connect Core 1.0 [133] The validation of the JWT the id token is encoded as, and the claims it contains. Validating a JWT correctly is critical, and errors in popular and commonly used libraries have occurred before [86]. As such in this project it is natural to use popular and vetted JWT libraries that can be relied upon. Unfortunately many of the libraries that would be suitable for this project did not fully support the `wasm32-wasi` compilation target. The following challenges where, and as of writing still is:

- Cryptographic libraries in Rust not supporting the `wasm32-wasi` compilation target, such as the *ring* library [145]. As ring is widely used in JWT and OpenID Connect libraries for Rust that narrowed the selection of reliable libraries to use in the thesis. Ring is working on support for the WASI interface, but as of writing the thesis the work is not yet complete [151].

- The WASM host implementation used by Envoy does not support *monotonic clocks* as specified in the WASI specification [171]. Monotonic clocks are useful when one wants to compare two instants against each other [150], like a timestamp in a JWT claim. The team behind the project was notified of the error which they acknowledged [41]. A fix is in progress, but will not be available in time for the thesis [168].

- The *proxy-wasm ABI* specification is currently under development, and new releases introduce major and potentially breaking changes. The WebAssembly Working Group is not in agreement on if the *proxy-wasm ABI* specification should be accepted as a standard as part of WASI or if other proposals covering similar use cases are better suited for standardization. The future of the *proxy-wasm ABI* specification is therefore somewhat uncertain [167].

Luckily a workaround for the JWT validation problem was figured out, first using *jwt-simple* as it supports the `wasm32-wasi` target, and after being made aware of the issue, the maintainers implemented workarounds for Envoy's monotonic clock problem. However due to practical concerns this limited the JWT signature support to RSA Signature with SHA-256 (RS256). RS256 is the most common JWT signature for API communications [90] so this did not really limit the work of the thesis.

Future of The Ecosystem

A programming language or platform is only as useful as the tooling and libraries that exists in the ecosystem surrounding it. As stated in the previous points WebAssembly has, at least in the Rust ecosystem some deficiencies. There is however work being done. Cryptography libraries take time to develop. The ring crate is working on supporting `wasm32-wasi` as a compilation target [151]. Additionally there is work being done in the WASI specification to include efficient implementation of cryptographic features in WebAssembly [160].

Chapter 6

Evaluation

> Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
>
> *Edsger W. Dijkstra, The Humble Programmer(1972)*

6.1 Functional Requirements

6.1.1 (R1) OAuth 2.0 and OpenID Connect

The *PoC filter* implements OAuth 2.0 with support for Authorization Code flow and OpenID Connect. The filter can be used for OAuth 2.0 delegated authorization without OpenID Connect. The user can specify if the filter should act as an OAuth 2.0 client or OpenID Connect client through the filter configuration. The JSON configuration in 6.1 configures the filter as a OpenID Connect client.

```
{
  "redirect_uri": "http://localhost:8090/callback",
  "auth_cluster": "cluster_mock_auth",
  "issuer": "http://mock-oauth2-server:8080/customiss",
  "token_uri": "http://mock-oauth2-server:8888/customiss/token",
  "auth_uri": "http://localhost:8888/customiss/authorize",
  "client_id": "myclientid",
  "client_secret": "myclientsecret",
  "scopes": ["openid", "email", "profile"],
  "cookie_expire": 120,
}
```

Source Code 6.1: Proof of Concept Filter Configuration

6.1.2 (R2) OAuth 2.0 - Authorization Code Flow with PKCE

The filter supports only Authorization Code Flow in compliance with the requirements from NAV's security platform team. The Authorization code flow is extended with PKCE. The source code 6.3 is the method in the filter source code responsible for crafting a redirect URL for a new Authorization Code flow. The PCKE code challenge is added to the URL and the verifier is saved for when the return redirect arrives.

```
pub fn authorization_url(&self, pkce_challenge: PkceCodeChallenge) -> (Url, CsrfToken) {
    let builder = self.client();
    let mut builder = builder
        .authorize_url(|| CsrfToken::new(util::new_random_verifier(32).secret().to_string()))
        // Set the PKCE code challenge.
        .set_pkce_challenge(pkce_challenge);

    // Add extra parameters for Authorization redirect from configuration
    for param in &self.extra_authorization_params {
        builder = builder.add_extra_param(param.0.as_str(), param.1.as_str());
    }

    // Add configured scopes
    for scope in &self.scopes {
        builder = builder.add_scope(Scope::new(scope.clone()))
    }

    builder.url()
}
```

Source Code 6.2: Proof of Concept Filter | `FilterConfig::authorization_url`

When the end user is redirected back from the authorization server the token request is crafted with the stored verifier. The token request is sent to the authorization server that completes the validation during the token request processing [103].

```
pub fn token_request(&self, code: String, code_verifier: Option<String>) -> HttpRequest {
    let mut params = vec![
        ("grant_type", "authorization_code"),
        ("code", code.as_str()),
    ];

    // if we have a PKCE verifer we send it in the token request
    let verifier_string;
    if code_verifier.is_some() {
        verifier_string = code_verifier.unwrap();
```

```
11          params.push(("code_verifier", verifier_string.as_str()))
12      }
13
14      util::token_request(
15          &AuthType::RequestBody,
16          &ClientId::new(self.client_id().to_string()),
17          Some(&ClientSecret::new(self.client_secret().to_string())),
18          &[],
19          Some(&RedirectUrl::from_url(self.redirect_uri.clone())),
20          None,
21          &TokenUrl::from_url(self.token_uri.clone()),
22          params,
23      )
24  }
```

Source Code 6.3: Proof of Concept Filter | `FilterConfig::token_request`

6.1.3 (R3) OpenID Connect Token Validation

The *Poc filter* implements token validation using features from the *jwt-simple* create, see section 5.1.4. The code in 6.4 is called when the filter receives a token back from the token request. *jwt-simple* is responsible for validating the JWT token signature, and JWT standard claims. Additionally after the JWT validation, the *azp* field is validated according to OpenID Connect specification. Note that validation step 7 and 8 are omitted. This is something that could be implemented as future work.

```
1   // Validates a OpenID Connect ID token
2   pub fn validate_token(&self, token: &str) -> Result<(), Error> {
3       let allowed_issuers: HashSet<String> =
4           vec![&self.issuer].iter().map(|s| s.to_string()).collect();
5
6       // Allowed audiences for ID token is client id and the issuer (for userinfo fetching)
7       let mut allowed_audiences = allowed_issuers.clone();
8       allowed_audiences.insert(self.client_id.clone());
9
10      let option = VerificationOptions {
11          reject_before: None,
12          accept_future: false,
13          required_subject: None,
14          required_key_id: None,
15          required_public_key: None,
16          required_nonce: None,
17          allowed_issuers: Some(allowed_issuers),
18          allowed_audiences: Some(allowed_audiences),
19          time_tolerance: None,
20          max_validity: None,
21      };
22      let claims = self.extra.validate_id_token(token, Some(option))?;
23      if let Some(jwt_simple::prelude::Audiences::AsSet(aud)) = claims.audiences {
24          // More than once one audience in token
25          if aud.len() > 1 {
```

```
26              // if azp claim is present, client id should be in azp field
27              match claims.custom.azp {
28                  None => {
29                      return Err(jwt_simple::Error::msg("azp field required"))
30                  }
31                  Some(azp) => {
32                      if azp != self.client_id {
33                          return Err(jwt_simple::Error::msg("azp is not valid"))
34                      }
35                  }
36              }
37          }
38      }
39      Ok(())
40  }
```

Source Code 6.4: Proof of Concept Filter | `FilterConfig::validate_token`

6.1.4 (R4) Configurable through the service mesh

The *PoC filter* is configurable through service mesh control plane configuration. The yaml configuration presented in Appendix A.3 shows an example configuration, note that a production configuration would use HTTPS. The `envoy.filters.http.wasm` name under `http_filters` defines the configuration of the WebAssembly Proof of Concept filter. The configuration is passed as embedded JSON data. The filter binary can be injected into the proxy as a file on disk, like in the example here, or delivered over the network using services like *WebAssemblyHub* [146].

6.2 Non-Functional Requirements

6.2.1 (R5) Performance

Test Environment

To test the performance of the filter implemented in the thesis a realistic, but scientific, setup needs to be created. The result of the performance test should be insight into how the filter would perform in regards to the metrics that developers and users value. The system and runtime used for the test is described in table 6.1 and 6.2. To make the test environment consistent and reproducible all components in the test environment are packaged as Docker containers, using Docker compose for test environment orchestration [20].

Test Configuration

The Docker Compose test environment is configured to create identical environments, except for the different alternatives. The configuration is available

Machine Specification	
Machine	MacBook Pro (15-inch, 2017)
Operating System	MacOS Big Sur 11.3.1
CPU	2,9 GHz Quad-Core Intel Core i7
RAM	16 GB 2133 MHz LPDDR3

Table 6.1: Test machine specification

Test Runtime Environment	
Runtime	Docker version 20.10.5, build 55c4c88
Orchestrator	Docker Compose
Resource Configuration	
Virtual CPU	4
Virtual RAM	6 Gigabyte

Table 6.2: Test runtime specification

in Appendix A.3. The performance test tool used is *Locust* [83]. The performance tests are defined as Locust Python scripts found in Appendix A.3. Locust describes itself as a:

> *Locust is an easy to use, scriptable and scalable performance testing tool. You define the behaviour of your users in regular Python code, instead of using a clunky UI or domain specific language. This makes Locust infinitely expandable and very developer friendly.* (Source: [83])

Locust Features

- **Python scripting:** Write test scenarios in plain old Python
- **Distributed and Scalable:** Supports simulating hundreds of thousands of users
- **GUI and CLI controllable**

Latency Benchmarking

The performance test used to evaluate the different approaches and how the PoC filter compares is a latency benchmark. It is important to note that the benchmark is not a full performance indicator of every configuration and workload. The test is meant to provide a as realistic as possible indication of performance. The intent of the benchmark is to show *out-of-the-box* configuration profiles without optimization, and use the proxies default configuration. The upstream backend is a simple echo service that returns the HTTP request back as the response.

Simulated Users	100
Duration	5 min
User Request Rate	.5/sec

Table 6.3: Locust benchmark latency test

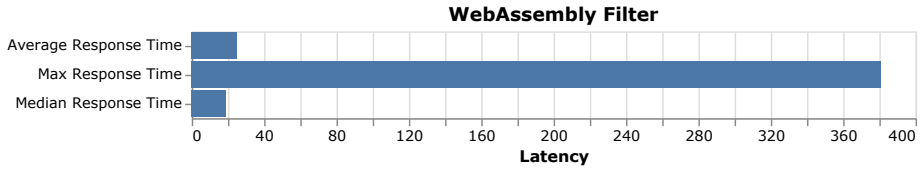

Figure 6.1: WebAssembly Filter Latency Statistics

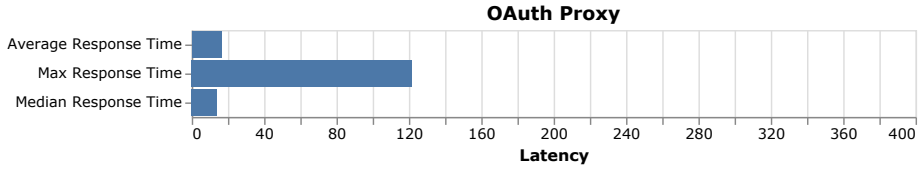

Figure 6.2: OAuth Proxy Latency Statistics

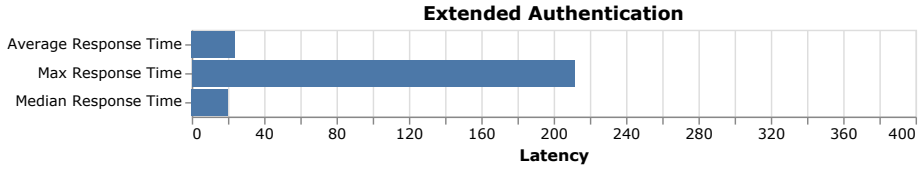

Figure 6.3: Extended Authorization Performance Statistics

Performance Test Results

The latency test results can be viewed in figure 6.1, 6.2, and 6.3. The metrics shown are **average response time**, **max response time**, and **median response time**. The unit latency is measured in **seconds**. Disregarding the max response time, the results are very comparable. The max response time is probably connected to the issues with the cache implementation discussed in section 5. Future development and testing is required.

6.2.2 (R6) DevOps

DevOps or platform teams need to provide *self-service* support to product teams using a hypothetical *PoC filter* powered service mesh. Figure 6.4 shows an example of this using Istio. Istio service mesh makes configuration and injecting of WebAssembly extensions possible using *EnvoyFilter* Custom Resource Definitions in Kubernetes (see 2.3.3).

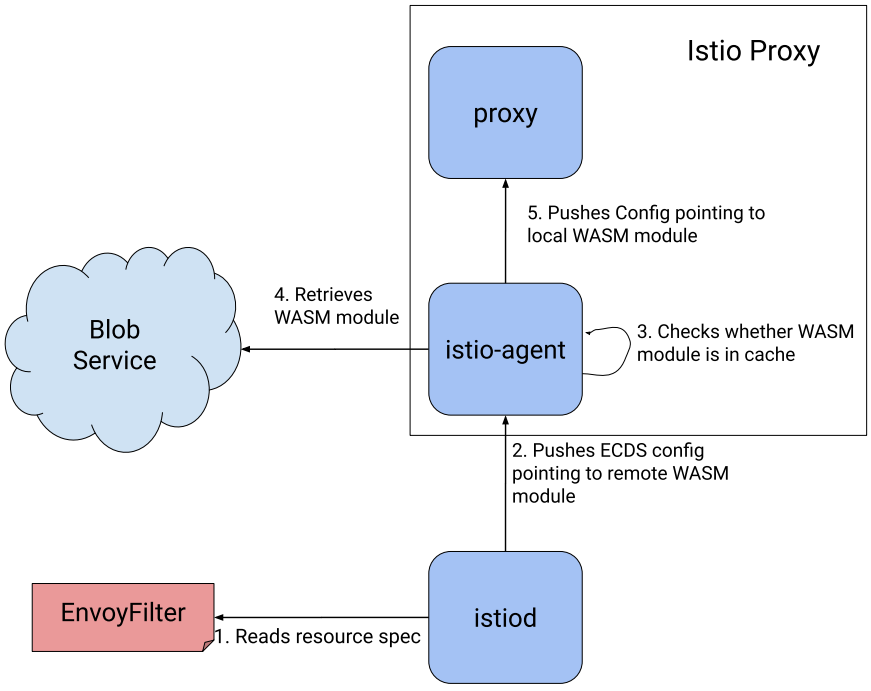

Figure 6.4: Diagram of WebAssembly module download and running in Istio (Source: [56])

1. Istio is configured by users through Kubernetes *Custom Resource Definitions (CRD)* [71] (see 2.3.3). When users apply a *EnvoyFilter* CRD to the cluster the service mesh daemon (istiod) reacts and kicks off the filter creation.

2. The daemon pushes configuration to istio-agents, daemons running on worker nodes in the cluster.

3. The istio-agents retrieve the WASM module from a storage service like *WebAssemblyHub* [146] or similar.

4. Finally istio-agent pushes the WebAssembly filter configuration to the service mesh data plane proxy. In the case of Envoy the configuration pushed would be similar to the example configuration presented in section 6.1.4

6.2.3 (R7) Security

Threat model The service mesh control plane, and the platform team administering it, is generally trusted in the case of evaluating the security regarding the *PoC filter*. In this threat model the upstream service is regarded as trusted. The focus is on untrusted downstream end-users or services, or other services inside the service mesh.

STRIDE

STRIDE is a threat modeling tool developed at Microsoft [136]. Following will be a STRIDE analysis of the *PoC filter* with respect to the two other approaches. The following STRIDE threat model analysis is not a complete enumeration of all possible threats, but rather a selection of what the thesis author considers to be the most relevant to discuss.

Spoofing The *PoC filter* provides better protection against spoofing attacks than the extended authorization or OAuth proxy approaches.

T1 **Attacker impersonates services authorization flow:** From the data plane proxy perspective *PoC filter* cannot be spoofed like external authorization or an external OAuth proxy could be. The risk of external services to be spoofed is limited by usage of security protocols like HTTPS with certificates that enables verification of service identity.

Tampering with data There is only a marginal difference in risk between the approaches due to commonly used network mitigations (HTTPS).

T2 **Attacker intercepts traffic inside the service mesh:** The *PoC filter* is communicating with the Envoy proxy at the process level so traffic between the filter and the proxy process cannot be intercepted over the network. When the *PoC filter* communicates with the authorization server, like during a token request, an attacker could engage in *Man-in-the-middle* attacks like stealing the security tokens. However, using OAuth 2.0 or OpenID Connect outside testing without HTTPS is not compliant with the respective specifications.

Repudiation for all approaches using HTTPS for communication is required to sufficiently protect against most non-repudiation attacks, additionally OAuth 2.0 is designed to mitigate non-repudiation attacks.

Information disclosure

T3 **Logging:** Authentication event logging can contain personal information (PII) or security critical data. If logging data is captured or stored in an insecure manner, or too much information is stored the system could be vulnerable to information disclosure attacks. The *PoC filter* logs events though the data plane proxy, meaning configuration of log levels and log capture is configured on the data plane proxy. With the other approaches there is another process emitting logs that need to be captured, stored, and indexed. *PoC filter* has lower risk of flawed logging setup leading to information disclosure.

Denial of service

T4 **Attacker attacks the service mesh with a Denial of Service attack, ex. DDoS:** Denial of Service attacks are usually mitigated against on ingress into the service mesh, using protections such as load balancer, Web Application Firewalls (WAF) [12], or provided by the cloud platform [37].

Elevation of privilege Elevation of privilege can be attempted through many different attack vectors. An attack vector that is most relevant for the *PoC filter* and the other approaches is *supply chain attacks* [95]. Attacks that attempt to gain access to software systems through compromising software the system relies on like build tools, third-party dependencies, or artifact storage. Attackers gaining control of a process responsible for authentication is critical and could result in damages such as stolen data, denial of service, unauthorized access, etc.

T5 **Attacker gains control over authentication process through supply chain attack:** The *PoC filter* is compiled to WebAssembly + WASI. Being a WebAssembly module means the filter gets the security benefits of WebAssembly's security model. The WebAssembly security model is designed to protect against potential malicious modules and support development by providing secure primitives. Modules run in a sandboxed environment and can only interact outside of the sandbox though imported APIs. WebAssembly modules execute deterministically, without undefined behaviour, with only a few exceptions [158]. Through the security features of WebAssembly modules are protected against attacks such as control flow hijacking, stack overflows, stack smashing, data execution, and most return-oriented programming (ROP) attacks [159]. These protections together with the import/export system, see 2.8, limits the

damage a malicious third-party dependency can do. If a module is compromised the behaviour of the module could potentially be changed, but due to the WebAssembly security model unlike other popular runtimes [113] arbitrary actions like accessing the file system or establishing network connection to external resources are much harder for an attacker to accomplish.

Chapter 7

Conclusion

7.1 Summary

In this thesis the goal has been to explore the token-based Zero Trust Architecture in the service mesh. Different approaches have been analysed and evaluated for suitability. A WebAssembly extension filter was selected as the best fit based on the requirements outlined in partnership with the thesis collaborator, Norwegian Labour and Welfare Administration (NAV). The filter, implementing a specified subset of OAuth 2.0 and OpenID Connect for a minimum viable product, was designed, implemented and tested according to performance, operational, and security criteria. The results of the filter evaluation, including functional and non-functional requirements, were positive compared to other approaches of extending the service mesh with authentication. The thesis results show that WebAssembly extensions are comparable to mature authentication projects in performance, are easy to operationalize using a service mesh control plane, and have strong security properties that make the technology suitable for a proxy context. The *Proof of Concept filter* was implemented using the Rust programming language, and WebAssembly + WASI, using the *proxy-wasm ABI* specification to integrate with service mesh proxies. However, the implementation phase revealed that the WebAssembly ecosystem is not quite yet ready for easy development of production level services, especially those needing high-level cryptographic libraries. Additionally several of the WebAssembly specifications are still in the volatile draft phase which could result in major breaking changes for the projects relying on them. Development teams that are prepared to write low-level functionality can still thrive though, as this thesis has shown. Finally, going forward extension of proxies, browsers and the cloud using WebAssembly seems very promising. The technology can, with some development, provide a powerful platform for Zero Trust Architecture implementation.

7.2 RQ1

What approaches exists for handling authentication and/or authorization in common service mesh implementations? In this thesis several options for how to implement authentication in a service mesh environment has been explored. The distinction and ultimately separation authentication from authorization was argued based on security considerations and technological development. Initially the option of implementing authentication directly in the application serviced by the service mesh was discussed, arguing that the approach introduces high risk of flawed implementation and redundant work. Further the thesis looked at different approaches to embedding authentication capabilities in the service mesh. Three different approaches that leveraged the service mesh infrastructure to different degrees were discussed;. implementing authentication as an extra proxy, through extended authorization API, or as WebAssembly extensions. Both the extra proxy approach and external authorization showed promising properties like mature implementations, low barrier of entry and well defined APIs. The drawbacks that are intrinsic to both are that they are standalone services resulting in latency, and potentially reducing reliability since they require extra network hops for downstream requests. Depending on configuration the approaches might also introduce a single point for failure. Finally the approach where the data plane proxy is extended with WebAssembly extensions where discussed, extending the proxy on the process level, benefiting from WebAssembly's security model. The main drawback for the extension filter approach is that the technology and surrounding ecosystem is relatively immature. As an example the *proxy-wasm ABI* specification and implementations in proxies is only months old at time of writing.

7.3 RQ2

What approach is best for performance in regards to latency, security capabilities and operational complexity for the serviced application? Using the service mesh allows for Zero Trust Architecture generally, and authentication specifically, to be provided as *Software-as-a-Service* products in an organisation. Allowing for consistent and efficient implementation of security services, while limiting bottle-necks in terms of performance or reliability. After the requirements phase the WebAssembly extension filter approach was considered to be the most promising. The Proof of concept filter was developed and evaluated favorably. The filter approach is service mesh native, and common service mesh implementations provide easy injection of custom filters today. WebAssembly modules include beneficial security features that provide protection against common application attacks. The extension filter is harder to spoof, or intercept in any way as it is not communicating to the proxy over a network connection, also lowering operational complexity and latency. There were issues with the WebAssembly extension ecosystem, primarily surrounding high-level, easy to use cryptographic libraries, for id token validation. The

ecosystem issue is a symptom of a new technology still developing, going forward the process level integration between the proxy and the filter should mean better and more reliable performance as the technology and libraries mature. The filter approach will benefit from the ongoing development effort to make WebAssembly performance as close to native as possible.

7.4 RQ3

How can the approach be implemented in a way that improves the security, while still keeping developers in control of the security, in line with cloud native development practices. Using Extension filter for the data plane proxy allows developer and operations to seamlessly and continuously configure and update the filter through the service mesh. The filter extensions allows for fast agile development as it can keep pace with a fast moving organization continuously deploying new services. However the question of how to best keep the developer “in the loop" has been a difficult question to answer. The thesis work has explored and discussed how Zero Trust Architecture can be implemented to create consistent security across services, but how to make that fit agile development values is hard. The current era of agile and the cloud native development promotes developer agency and creating cross-functional teams that can take ownership of everything from the business domain to the software, there seems to be a fundamental trade-off between developer agency and consistent security implementation for organisations. Future work is needed, researching agile security and Zero Trust Architecture.

7.5 Future Work

Identity and authentication is fundamental for communication in today's internet based culture, and how that communication is critical for trust in our society. Zero Trust Architecture and the ideas it espouses represents a paradigm shift in how organisations can secure the services that power the world. This thesis has shown how service mesh technology can be leveraged to implement Zero Trust Architecture providing a path forward for more secure platforms native to the cloud era. In regards to WebAssembly, this thesis has discussed just a few of its use cases, other domains, like IoT or edge computing can prove to be even more powerful, WebAssembly is posed to become an important part of the software industry going forward. This thesis has focused on authentication, but an avenue for service mesh technology might be end-user access control, as this area is also crowded with vulnerabilities. Finally this thesis has hopefully revealed the potential of zero trust architectures on the service mesh. Further work in incorporating more OAuth 2.0 specifications such as *OAuth 2.0 Token Exchange* and *OAuth 2.0 Dynamic Client Registration Protocol* are needed to make WebAssembly extensions in the service mesh a production ready approach to token-based Zero Trust Architecture.

Bibliography

[1] 451 Research. *Observability and Security Are Important Service Mesh Benefits approval of S&P Global Market Intelligence. Service Mesh Adoption*. Tech. rep. 2020. URL: https://aspenmesh.io/what-are-companies-using-service-mesh-for/.

[2] Martín Abadi et al. *Control-Flow Integrity Principles, Implementations, and Applications*. Tech. rep. Nov. 2005. URL: https://www.microsoft.com/en-us/research/publication/control-flow-integrity/.

[3] Saleh Alshomrani and Shahzad Qamar. 'Cloud Based E-Government: Benefits and Challenges'. In: *INTERNATIONAL JOURNAL OF MULTIDISCIPLINARY SCIENCES AND ENGINEERING* 4.6 (2013). ISSN: 2045-7057. URL: www.ijmse.org.

[4] *Awesome WebAssembly Languages*. Date accessed: 24/03/2021. URL: https://github.com/appcypher/awesome-wasm-langs.

[5] Barracuda Networks. *What is a Network Perimeter?* Date accessed: 23/05/2021. URL: https://www.barracuda.com/glossary/network-perimeter.

[6] Evan Barth and Doug Gilman. *Zero Trust Networks*. O'Reilly Media, Inc., 2017.

[7] Betsy Beyer et al. *Site reliability engineering: how Google runs production systems*. OReilly, 2016.

[8] Business Wire. *Global Authentication Services Market Analysis, Trends, and Forecasts 2019-2025*. Date accessed: 16/05/2021. URL: https://www.businesswire.com/news/home/20200102005247/en/Global-Authentication-Services-Market-Analysis-Trends-and-Forecasts-2019-2025---ResearchAndMarkets.com.

[9] Bytecode Alliance. *Experimental WASI API bindings for Rust*. Date accessed: 17/04/2021. URL: https://github.com/bytecodealliance/wasi.

[10] Bytecode Alliance. *Testing in WASI - The cargo-wasi Subcommand*. Date accessed: 19/05/2021. URL: https://bytecodealliance.github.io/cargo-wasi/testing.html.

[11] Bytecode Alliance. *Wasmtime — a small and efficient runtime for WebAssembly & WASI*. Date accessed: 24/03/2021. URL: https://wasmtime.dev/.

[12] Alfred Chung. *Application Layer Protection for Istio Service Mesh -*. Date accessed: 24/05/2021. URL: https://www.signalsciences.com/blog/cloud-native-protection-istio-service-mesh/.

[13] Cloud Native Computing Foundation. *Cloud Native Computing Foundation ("CNCF") Charter*. Date accessed: 05/05/2021. URL: https://github.com/cncf/foundation/blob/master/charter.md.

[14] Alex Crichton and David Ramos. *oauth2 - Rust*. Date accessed: 19/05/2021. URL: https://docs.rs/oauth2/4.0.0/oauth2/.

[15] Curity. *Zero Trust Architecture is a Token-Based Architecture*. Date accessed: 24/05/2021. URL: https://curity.io/resources/learn/zero-trust-overview/.

[16] D. Hardt Ed. *RFC 6749 - The OAuth 2.0 Authorization Framework*. Tech. rep. Internet Engineering Task Force, 2012. URL: https://tools.ietf.org/html/rfc6749.

[17] Frank Denis. *jwt_simple - Rust*. Date accessed: 19/05/2021. URL: https://docs.rs/jwt-simple/0.10.0/jwt_simple/.

[18] Dillon Gerred. *Benchmarking 5 Popular Load Balancers: Nginx, HAProxy, Envoy, Traefik, and ALB | Loggly*. Date accessed: 09/03/2021. URL: https://www.loggly.com/blog/benchmarking-5-popular-load-balancers-nginx-haproxy-envoy-traefik-and-alb/.

[19] Cem Dilmegani, Bengi Korkmaz, and Martin Lundqvist. *Public-sector digitization: The trillion-dollar challenge | McKinsey*. Date accessed: 31/05/2020. URL: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/public-sector-digitization-the-trillion-dollar-challenge.

[20] Docker Inc. *Docker Documentation*. Date accessed: 12/05/2021. URL: https://docs.docker.com/.

[21] Docker Inc. *Overview of Docker Compose | Docker Documentation*. Date accessed: 07/05/2021. URL: https://docs.docker.com/compose/.

[22] Bill Doerrfeld. *Extending the Envoy Proxy With WebAssembly*. Date accessed: 17/05/2021. URL: https://containerjournal.com/features/extending-the-envoy-proxy-with-webassembly/.

[23] Sébastien Dubois. *Why you shouldn't implement authorization "around" your application and what to do instead*. Date accessed: 26/05/2021. URL: https://itnext.io/why-you-shouldnt-implement-authorization-around-your-application-and-what-to-do-instead-35c7851bfd9f.

[24] Melvin E. Conway. *How Do Committees Invent?* Tech. rep. Date accessed: 22/05/2021: F. D. Thompson Publications, Inc., 1968. URL: http://www.melconway.com/research/committees.html.

[25] Elkjøp. *A $4 billion retail giant built Kubernetes platform powered by Linkerd*. Date accessed: 05/05/2021, 2021. URL: https://buoyant.io/case-studies/elkjop/.

[26] Envoy. *Envoy Proxy - Home*. Date accessed: 09/03/2021. URL: https://www.envoyproxy.io/.

[27] Envoy. *envoyproxy/envoy: Cloud-native high-performance edge/middle/service proxy*. Date accessed: 09/03/2021. URL: https://github.com/envoyproxy/envoy.

[28] Envoy. *External Authorization*. Date accessed: 06/05/2021. URL: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/security/ext_authz_filter.

[29] Envoy. *Router Filter*. Date accessed: 24/05/2021. URL: https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/router_filter.

[30] Envoy Project Authors. *Life of a Request*. Date accessed: 17/05/2021. URL: https://www.envoyproxy.io/docs/envoy/latest/intro/life_of_a_request.

[31] Facebook. *Introducing Proxygen, Facebook's C++ HTTP framework*. Date accessed: 15/05/2021. URL: https://engineering.fb.com/2014/11/05/production-engineering/introducing-proxygen-facebook-s-c-http-framework/.

[32] Daniel Fett, Ralf Kuesters, and Guido Schmitz. 'A Comprehensive Formal Security Analysis of OAuth 2.0'. In: *Proceedings of the ACM Conference on Computer and Communications Security* 24-28-October-2016 (Jan. 2016), pp. 1204–1215. URL: http://arxiv.org/abs/1601.01229.

[33] Roy Thomas Fielding and Richard N Taylor. 'Architectural Styles and the Design of Network-Based Software Architectures'. PhD thesis. 2000. ISBN: 0599871180.

[34] Flexera. *RightScale 2019 State of the Cloud Report from Flexera*. Tech. rep. 2019. URL: https://resources.flexera.com/web/media/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf.

[35] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution, 2018.

[36] Evan Gilman and Doug Barth. *Zero trust networks : building secure systems in untrusted networks*. O'Reilly Media, 2017, p. 223. ISBN: 9781491962190.

[37] Google Cloud. *Cloud Armor Network Security | Google Cloud Armor*. Date accessed: 24/05/2021. URL: https://cloud.google.com/armor.

[38] Google Cloud Platform. *Cloud SQL proxy client and Go library*. Date accessed: 13/05/2021. URL: https://github.com/GoogleCloudPlatform/cloudsql-proxy.

[39] Oliver Gould. *The road ahead for Linkerd2-proxy, and how you can get involved*. Date accessed: 10/03/2021. URL: https://linkerd.io/2020/09/02/the-road-ahead-for-linkerd2-proxy/.

[40] Pankaj Gupta. *Mutual TLS: Securing Microservices in Service Mesh*. Date accessed: 15/05/2021. URL: https://thenewstack.io/mutual-tls-microservices-encryption-for-service-mesh/.

[41] Sondre Halvorsen. *Issue with monotonic clocks · Issue #103 · proxy-wasm/proxy-wasm-rust-sdk*. Date accessed: 26/04/2021. URL: https://github.com/proxy-wasm/proxy-wasm-rust-sdk/issues/103.

[42] D Hardt et al. *The OAuth 2.1 Authorization Framework draft-ietf-oauth-v2-1-02*. Date accessed: 22/03/2021, 2021. URL: https://tools.ietf.org/html/draft-ietf-oauth-v2-1-02.

[43] Yaron Haviv. *KubeCon 2018: Kubernetes is the New OS, So What?* Date accessed: 23/05/2021. URL: https://thenewstack.io/kubecon-2018-kubernetes-is-the-new-os-so-what/.

[44] Herman David, Wagner Luke, and Zakai Alon. *asm.js | Working Draft*. Date accessed: 24/03/2021, 2014. URL: http://asmjs.org/spec/latest/.

[45] Heroku. *Cloud Application Platform | Heroku*. Date accessed: 12/05/2021. URL: https://www.heroku.com/.

[46] Pat Hickey et al. 'WebAssembly/WASI'. In: (Dec. 2020). DOI: 10.5281/ZENODO.4323447. URL: https://doi.org/10.5281/zenodo.4323447#.YFyFyY9oplo.mendeley.

[47] Kevin Hoffman. *Programming WebAssembly with Rust: unified development for web, mobile, and embedded applications*. The Pragmatic Programmers., 2019.

[48] https://linuxcontainers.org/. *Linux Containers - LXC - Introduction*. Date accessed: 12/05/2021. URL: https://linuxcontainers.org/lxc/introduction/.

[49] Solomon Hykes. *Solomon Hykes on Twitter*. Date accessed: 22/05/2021. URL: https://twitter.com/solomonstre/status/1111004913222324225.

[50] Nick Hynes. *jsonwebkey - Rust*. Date accessed: 19/05/2021. URL: https://docs.rs/jsonwebkey/0.3.2/jsonwebkey/.

[51] Internet Engineering Task Force. *IETF | RFCs*. Date accessed: 22/03/2021. URL: https://www.ietf.org/standards/rfcs/.

[52] Istio. *Authservice*. Date accessed: 17/04/2021. URL: https://github.com/istio-ecosystem/authservice.

[53] Istio. *Istio / Architecture*. Date accessed: 17/05/2021. URL: https://istio.io/latest/docs/ops/deployment/architecture/.

[54] Istio. *Istio / Security*. Date accessed: 05/05/2021. URL: https://istio.io/latest/docs/concepts/security/.

[55] Istio. *Istio / What is Istio?* Date accessed: 08/03/2021. URL: https://istio.io/latest/docs/concepts/what-is-istio/.

[56] Istio. *Istio and Envoy WebAssembly Extensibility, One Year On*. Date accessed: 22/05/2021. URL: https://istio.io/latest/blog/2021/wasm-progress/.

[57] Istio. *Mixer*. Date accessed: 13/05/2021. URL: https://istio-releases.github.io/v0.1/docs/concepts/policy-and-control/mixer.html.

[58] Istio. *Redefining extensibility in proxies - introducing WebAssembly to Envoy and Istio*. Date accessed: 13/05/2021. URL: https://istio.io/latest/blog/2020/wasm-announce/.

[59] M Jones et al. *RFC 7515 - JSON Web Signature (JWS)*. 2015. URL: https://tools.ietf.org/html/rfc7515.

[60] M Jones et al. *RFC 7519 - JSON Web Token (JWT)*. Tech. rep. Internet Engineering Task Force, 2015. URL: https://tools.ietf.org/html/rfc7519.

[61] M. Jones et al. *OAuth 2.0 Token Exchange*. Tech. rep. Internet Engineering Task Force (IETF), 2020. URL: https://datatracker.ietf.org/doc/html/rfc8693.

[62] M. Jones et al. *rfc7523 - JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants*. Tech. rep. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7523.

[63] M. Jones et al. *rfc7591 - OAuth 2.0 Dynamic Client Registration Protocol*. Tech. rep. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7591.

[64] Poul-Henning Kamp and Robert N M Watson. *Jails: Confining the omnipotent root*. Tech. rep. The FreeBSD Project, 2000. URL: https://papers.freebsd.org/2000/phk-jails/.

[65] Michael Kerrisk. *Namespaces in operation, part 1: namespaces overview*. Date accessed: 12/05/2021, 2013. URL: https://lwn.net/Articles/531114/.

[66] Gene Kim et al. *The DevOps handbook: how to create world-class agility, reliability, and security in technology organizations*. IT Revolution Press, LLC, 2017.

[67] John Kindervag. *No More Chewy Centers: Introducing The Zero Trust Model Of Information Security*. Tech. rep. Forrester, 2010. URL: www.forrester.com..

[68] Jack Kleeman. *1500 microservices at @monzo*. Date accessed: 05/05/2021. URL: https://twitter.com/JackKleeman/status/1190354757308862468.

[69] Kubernetes. *Operator pattern*. Date accessed: 23/05/2021. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[70] Kubernetes. *Pods | Kubernetes*. Date accessed: 08/03/2021. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[71] Kubernetes. *Use Custom Resources*. Date accessed: 22/05/2021. URL: https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/_print/#pg-dc64883f1fd119402b112d3ff6733452.

[72] Kubernetes. *What is Kubernetes? | Kubernetes*. Date accessed: 08/03/2021. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[73] Patrick Lacharme et al. *The Linux Pseudorandom Number Generator Revisited*. Tech. rep. URL: https://eprint.iacr.org/2012/251.pdf.

[74] Kin Lane. *Intro to APIs: History of APIs*. Date accessed: 22/05/2021. URL: https://blog.postman.com/intro-to-apis-history-of-apis/.

[75] Paul J. Leach et al. 'Hypertext Transfer Protocol – HTTP/1.1'. In: *The Internet Society* (1999). URL: https://tools.ietf.org/html/rfc2616.

[76] James Lewis and Martin Fowler. *Microservices*. Date accessed: 31/05/2020, 2014. URL: https://martinfowler.com/articles/microservices.html.

[77] James Lewis and Martin Fowler. *Microservices*. Date accessed: 05/05/2021. URL: https://martinfowler.com/articles/microservices.html.

[78] Wubin Li et al. 'Service Mesh: Challenges, state of the art, and future research opportunities'. In: *Proceedings - 13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, 10th International Workshop on Joint Cloud Computing, JCC 2019 and 2019 IEEE International Workshop on Cloud Computing in Robotic Systems, CCRS 2019*. Institute of Electrical and Electronics Engineers Inc., May 2019, pp. 122–127. ISBN: 9781728114415. DOI: 10.1109/SOSE.2019.00026.

[79] Tim Lindholm et al. *The Java ® Virtual Machine Specification Java SE 15 Edition*. Tech. rep. 2020.

[80] Linkerd. *Automatic mTLS*. Date accessed: 23/05/2021. URL: https://linkerd.io/2.10/features/automatic-mtls/.

[81] Linkerd. *linkerd/linkerd2-proxy: A purpose-built proxy for the Linkerd service mesh. Written in Rust.* Date accessed: 09/03/2021. URL: https://github.com/linkerd/linkerd2-proxy.

[82] Linkerd. *Overview | Linkerd*. Date accessed: 31/05/2020. URL: https://linkerd.io/2/overview/.

[83] locustio. *locustio/locust: Scalable user load testing tool written in Python*. Date accessed: 19/05/2021. URL: https://github.com/locustio/locust.

[84] Lodderstedt T. et al. *OAuth 2.0 Security Best Current Practice draft-ietf-oauth-security-topics-16*. Date accessed: 23/03/2021, 2020. URL: https://tools.ietf.org/html/draft-ietf-oauth-security-topics-16.

[85] Neil Madden. *API security in action*. Manning, 2020.

[86] Tim McLean. *Critical vulnerabilities in JSON Web Token libraries*. Date accessed: 26/04/2021. URL: https://www.chosenplaintext.ca/2015/03/31/jwt-algorithm-confusion.html.

[87] MDN. *Understanding WebAssembly text format*. Date accessed: 25/05/2021. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format.

[88] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology*. Tech. rep. DOI: 10.6028/NIST.SP.800-145.

[89] Paul Menage. *CGROUPS*. Tech. rep. Date accessed: 12/05/2021: Linux Kernel Organization, Inc. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[90] Shawn Meyer. *Navigating RS256 and JWKS*. Date accessed: 25/05/2021. URL: https://auth0.com/blog/navigating-rs256-and-jwks/.

[91] Microsoft. *Common Language Runtime (CLR) overview - .NET*. Date accessed: 24/03/2021. URL: https://docs.microsoft.com/en-us/dotnet/standard/clr.

[92] Microsoft. *Defining Cloud Native*. Date accessed: 05/05/2021. URL: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition.

[93] Microsoft. *Microsoft identity platform and OAuth2.0 On-Behalf-Of flow*. Date accessed: 23/05/2021. URL: https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-on-behalf-of-flow.

[94] Microsoft Docs. *Data Execution Prevention*. Date accessed: 25/05/2021. URL: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[95] Microsoft Docs. *Supply chain attacks - Windows security*. Date accessed: 24/05/2021. URL: https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/supply-chain-malware.

[96] Microsoft Docs. *Windows API index - Win32 apps*. Date accessed: 23/05/2021. URL: https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list.

[97] Sammy Migues, John Steven, and Mike Ware. *BSIMM11*. Tech. rep. 2020, pp. 0–114. URL: https://www.bsimm.com/download.html.

[98] George Miranda. *The Service Mesh*. O'Reilly Media, Inc., 2018.

[99] William Morgan. *The History of the Service Mesh*. Date accessed: 15/05/2021. URL: https://thenewstack.io/history-service-mesh/.

[100] Chip Morningstar. *What Are Capabilities?* Date accessed: 23/05/2021. URL: http://habitatchronicles.com/2017/05/what-are-capabilities/.

[101] Mozilla. *Authorization - HTTP | MDN*. Date accessed: 17/04/2021. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization.

[102] Mozilla. *X-Forwarded-For - HTTP | MDN*. Date accessed: 23/03/2021. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-For.

[103] Ed. N. Sakimura et al. *rfc7636 | Proof Key for Code Exchange by OAuth Public Clients*. Tech. rep. URL: https://datatracker.ietf.org/doc/html/rfc7636.

[104] N. Sakimura et al. *Final: OpenID Connect Discovery 1.0 incorporating errata set 1*. Tech. rep. Date accessed: 09/05/2021: The OpenID Foundation, 2014. URL: https://openid.net/specs/openid-connect-discovery-1_0.html.

[105] N. Sakimura Ed. et al. *RFC 7636 - Proof Key for Code Exchange by OAuth Public Clients*. Tech. rep. Internet Engineering Task Force, 2015. URL: https://tools.ietf.org/html/rfc7636.

[106] *nais.io*. Date accessed: 24/05/2021. URL: https://nais.io/.

[107] National Institute of Standards and Technology. 'Zero Trust Architecture - Draft (2nd) NIST Special Publication 800-207'. 2020. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207-draft2.pdf.

[108] NAV. *Antall deployments av applikasjoner i NAV*. Date accessed: 05/05/2021. URL: https://data.nav.no/datapakke/e1556a04a484bbe06dda2f6b874f3dc1.

[109] Netflix. *Hystrix*. Date accessed: 15/05/2021. URL: https://github.com/Netflix/hystrix.

[110] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., Feb. 2015, p. 102. ISBN: 9781491950357.

[111] OAuth2 Proxy. *OAuth2 Proxy*. Date accessed: 17/04/2021. URL: https://oauth2-proxy.github.io/oauth2-proxy/.

[112] OAuth2 Proxy. *Overview | OAuth2 Proxy*. Date accessed: 25/05/2021. URL: https://oauth2-proxy.github.io/oauth2-proxy/docs/configuration/overview.

[113] Andres Ojamaa and Karl Düüna. 'Assessing the security of Node.js platform'. In: *2012 International Conference for Internet Technology and Secured Transactions*. Dec. 2012, pp. 348–355.

[114] Okta Inc. *Getting Started with Zero Trust Never trust, always verify*. Tech. rep. 2020. URL: https://www.okta.com/uk/resources/whitepaper-zero-trust-with-okta-modern-approach-to-secure-access/.

[115] Hans Olav Mugås. *Foreldrepengeprosjektet i NAV vant Digitaliseringsprisen 2019*. Date accessed: 23/05/2021. URL: https://memu.no/artikler/foreldrepengeprosjektet-i-nav-vant-digitaliseringsprisen-2019/.

[116] Open Container Initiative. *opencontainers/image-spec*. Date accessed: 12/05/2021. URL: https://github.com/opencontainers/image-spec/blob/master/spec.md.

[117] Open Policy Agent. *Open Policy Agent | Introduction*. Date accessed: 16/05/2021. URL: https://www.openpolicyagent.org/docs/latest/.

[118] OpenID. *OpenID Connect | OpenID*. Date accessed: 22/03/2021. URL: https://openid.net/connect/.

[119] Oracle. *JAR File Specification*. Date accessed: 12/05/2021. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html.

[120] Barclay Osborn et al. 'BeyondCorp: Design to Deployment at Google'. In: (2016). URL: https://www.usenix.org/publications/login/spring2016/osborn.

[121] *OWASP Threat Modeling Playbook (OTMP)*. URL: https://owasp.org/www-project-threat-modeling-playbook/.

[122] Mathias Payer. 'Control-Flow Hijacking: Are We Making Progress?' In: May 2017, p. 4. DOI: 10.1145/3052973.3056127.

[123] L Peter Deutsch. *The Eight Fallacies of Distributed Computing*. Date accessed: 15/05/2021. URL: https://web.archive.org/web/20160909234753/https://blogs.oracle.com/jag/resource/Fallacies.html.

[124] PortSwigger. *OAuth 2.0 authentication vulnerabilities*. Date accessed: 26/05/2021. URL: https://portswigger.net/web-security/oauth#exploiting-oauth-authentication-vulnerabilities.

[125] Christian Posta. *The Hardest Part of Microservices: Calling Your Services*. Date accessed: 10/03/2021. URL: https://www.slideshare.net/ceposta/the-hardest-part-of-microservices-calling-your-services.

[126] Daniel Price and Andrew Tucker. *Solaris Zones: Operating System Support for Consolidating Commercial Workloads*. Tech. rep. Sun Microsystems, Inc., 2004. URL: https://www.usenix.org/legacy/publications/library/proceedings/lisa04/tech/full_papers/price/price.pdf.

[127] Puppet. *State of DevOps Report 2020 | presented by Puppet and CircleCI*. Tech. rep. 2021.

[128] Tasdik Rahman. *Handling signals for applications running in kubernetes*. Date accessed: 12/05/2021. URL: https://tasdikrahman.me/2019/04/24/handling-singals-for-applications-in-kubernetes-docker/.

[129] Matt Raible. *What the Heck is OAuth? | Okta Developer*. Date accessed: 21/03/2021. URL: https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth.

[130] redis.io. *Redis*. Date accessed: 25/05/2021. URL: https://redis.io/.

[131] *Rust Programming Language*. Date accesed: 18/05/2021. URL: https://www.rust-lang.org/.

[132] N Sakimura et al. *Draft: OpenID Connect Basic Client Implementer's Guide 1.0 - draft 40*. Date accessed: 18/04/2021, May 2020. URL: https://openid.net/specs/openid-connect-basic-1_0.html.

[133] Sakimura N. et al. *Final: OpenID Connect Core 1.0 incorporating errata set 1*. Tech. rep. Date accessed: 22/03/2021, 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html.

[134] Kamal Sheel Mishra and Anil Kumar Tripathi. *Some Issues, Challenges and Problems of Distributed Software System*. Tech. rep. Varanasi: Indian Institute of Technology, 2014, pp. 4922–4925. URL: www.ijcsit.com.

[135] Adam Shostack. *Elevation of Privilege: Drawing Developers into Threat Modeling*. Tech. rep. 2014.

[136] Adam Shostack. *Threat modeling : designing for security*. Indianapolis, IN: John Wiley and Sons, 2014. ISBN: 9781118809990.

[137] Huzaifa Sidhpurwala. *Stack Smashing Protection (StackGuard)*. Date accessed: 25/05/2021. URL: https://access.redhat.com/blogs/766093/posts/3548631.

[138] Piotr Sikora. *WebAssembly for Proxies (Rust SDK)*. Date accessed: 09/05/2021. URL: https://github.com/proxy-wasm/proxy-wasm-rust-sdk.

[139] Piotr Sikora. *WebAssembly in Envoy*. Date accessed: 10/03/2021. URL: https://github.com/proxy-wasm/spec/blob/master/docs/WebAssembly-in-Envoy.md.

[140] Piotr Sikora and Takeshi Yoneda. *WebAssembly for Proxies (ABI specification)*. Date accessed: 15/05/2021. URL: https://github.com/proxy-wasm/spec.

[141] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Operating System Concepts, 9th Edition*. John Wiley & Sons, 2012.

[142] Matthew Skelton and Manuel Pais. *Key Concepts — Team Topologies*. Date accessed: 22/05/2021. URL: https://teamtopologies.com/key-concepts.

[143] Matthew Skelton and Manuel Pais. *What is a Thinnest Viable Platform (TVP)?* Date accessed: 16/05/2021. URL: https://teamtopologies.com/key-concepts-content/what-is-a-thinnest-viable-platform-tvp.

[144] Richard Skowyra et al. 'Systematic analysis of defenses against return-oriented programming'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8145 LNCS. Springer, Berlin, Heidelberg, Oct. 2013, pp. 82–102. ISBN: 9783642412837. DOI: 10.1007/978-3-642-41284-4{_}5. URL: https://link-springer-com.ezproxy.uio.no/chapter/10.1007/978-3-642-41284-4_5.

[145] Brian Smith. *ring: Safe, fast, small crypto using Rust*. Date accessed: 26/04/2021. URL: https://github.com/briansmith/ring.

[146] Solo.io. *WASM Hub*. Date accessed: 17/04/2021. URL: https://webassemblyhub.io/.

[147] Travis Spencer. *(4) GOTO 2018 • Securing APIs and Microservices with OAuth and OpenID Connect • Travis Spencer - YouTube*. Date accessed: 21/03/2021, 2019. URL: https://www.youtube.com/watch?v=hTgff3cJ6AU.

[148] SPIFFE. *SPIFFE Overview*. Date accessed: 06/05/2021. URL: https://spiffe.io/docs/latest/spiffe-about/overview/.

[149] Stackrox. *State of Container and Kubernetes Security*. Tech. rep. 2020. URL: https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers.

[150] *std::time::Instant - Rust*. Date accessed: 26/04/2021. URL: https://doc.rust-lang.org/std/time/struct.Instant.html.

[151] *Support WASI by nhynes · Pull Request #900 · briansmith/ring*. Date accessed: 26/04/2021. URL: https://github.com/briansmith/ring/pull/900.

[152] The Jaeger Authors. *Jaeger*. Date accessed: 15/05/2021. URL: https://www.jaegertracing.io/docs/.

[153] The National Security Agency (NSA). *Embracing a Zero Trust Security Model*. Tech. rep. Date accessed: 25/05/2021, 2021. URL: https://media.defense.gov/2021/Feb/25/2002588479/-1/-1/0/CSI_EMBRACING_ZT_SECURITY_MODEL_UOO115131-21.PDF.

[154] The Norwegian Digitalisation Agency. *ID-porten | Digdir*. Date accessed: 21/05/2021. URL: https://www.digdir.no/digitale-felleslosninger/id-porten/864.

[155] The Norwegian Digitalisation Agency. *Maskinporten | Digdir*. Date accessed: 21/05/2021. URL: https://www.digdir.no/digitale-felleslosninger/maskinporten/869.

[156] The Rust Programming Language. *Introduction - The Cargo Book*. Date accessed: 18/05/2021. URL: https://doc.rust-lang.org/cargo/index.html.

[157] The Rust Programming Language. *Platform Support - The rustc book*. Date accessed: 19/05/2021. URL: https://doc.rust-lang.org/nightly/rustc/platform-support.html.

[158] The WebAssembly Community Group. *Nondeterminism in WebAssembly*. Date accessed: 24/05/2021. URL: https://github.com/WebAssembly/design/blob/main/Nondeterminism.md.

[159] The WebAssembly Community Group. *Security - WebAssembly*. Date accessed: 24/05/2021. URL: https://webassembly.org/docs/security/.

[160] The WebAssembly Community Group. *WASI Cryptography APIs*. Date accessed: 25/05/2021. URL: https://github.com/WebAssembly/wasi-crypto.

[161] Seth Thompson. *Experimental support for WebAssembly in V8*. Date accessed: 24/03/2021. URL: https://v8.dev/blog/webassembly-experimental.

[162] Tommy Trøen et al. *nais/tokendings*. Date accessed: 24/05/2021. URL: https://github.com/nais/tokendings.

[163] Twitter. *Finagle*. Date accessed: 15/05/2021. URL: https://twitter.github.io/finagle/.

[164] W3C WebAssembly Commmunity Group. *Introduction — WebAssembly 1.1*. Date accessed: 25/03/2021. URL: https://webassembly.github.io/spec/core/intro/introduction.html.

[165] W3C WebAssembly Commmunity Group. *Overview — WebAssembly 1.1*. Date accessed: 25/03/2021. URL: https://webassembly.github.io/spec/core/intro/overview.html.

[166] Stephen R. Walli. 'The POSIX family of standards'. In: *StandardView* 3.1 (Mar. 1995), pp. 11–17. ISSN: 1067-9936. DOI: 10.1145/210308.210315. URL: https://dl.acm.org/doi/abs/10.1145/210308.210315.

[167] WASI Subgroup. *Agenda for the April 22 video call of WASI Subgroup*. Date accessed: 25/05/2021. URL: https://github.com/WebAssembly/WASI/blob/linclark-patch-6/meetings/2021/WASI-04-22.md.

[168] *wasi: support monotonic clock on clock_time_get. by mathetake · Pull Request #156 · proxy-wasm/proxy-wasm-cpp-host*. Date accessed: 26/04/2021. URL: https://github.com/proxy-wasm/proxy-wasm-cpp-host/pull/156.

[169] WebAssembly. *WebAssembly High-Level Goals - WebAssembly*. Date accessed: 10/03/2021. URL: https://webassembly.org/docs/high-level-goals/.

[170] *WebAssembly Community Group*. Date accessed: 24/03/2021. URL: https://www.w3.org/community/webassembly/.

[171] WebAssembly Working Group. *WASI API*. Date accessed: 24/03/2021. URL: https://github.com/WebAssembly/WASI/blob/main/phases/snapshot/docs.md.

[172] WebAssembly Working Group. *WebAssembly Web API*. Date accessed: 24/03/2021. URL: https://webassembly.github.io/spec/web-api/index.html.

[173] *Why Software Is Eating the World - Andreessen Horowitz*. Date accessed: 06/02/2021. URL: https://a16z.com/2011/08/20/why-software-is-eating-the-world/.

[174] Dave Wichers and Jeff Williams. 'Owasp top-10 2017'. In: *OWASP Foundation* (2017). URL: https://owasp.org/www-project-top-ten/2017/.

[175] Adam Wiggins. *The Twelve-Factor App*. Date accessed: 12/05/2021. URL: https://12factor.net/.

Appendices

Appendix A

Proof of Concept Filter

A.1 Source Code

https://github.com/sonhal/wasm-oauth-filter/

A.2 Dependencies

https://github.com/sonhal/wasm-oauth-filter/blob/master/Cargo.toml

A.3 Performance Test Suite

https://github.com/sonhal/wasm-oauth-filter/tree/master/perf-test

A.4 Envoy Filter Configuration

https://github.com/sonhal/wasm-oauth-filter/blob/master/test-env/envoy-bootstrap-static-loading.yaml