POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master Thesis

Zero Trust networks with Istio

Supervisors
Prof. Antonio Lioy
Dott. Ignazio Pedone

Candidate
Matteo Pace

Academic Year 2020-2021

Contents

Chapter 1

Introduction

Cloud computing has drastically changed the approach of deploying and accessing data and services, increasing the availability, flexibility and evolving the investment model by reducing initial costs and associated risks. The transitioning into cloud solutions has become even more accelerated during the COVID-19 pandemic, striving for resilience, flexible computational power demand and activities more than ever migrated into the online space[1].
Even the virtualized deployment approach made of virtual machines, already better in terms of utilization of resources and scalability in respect to the traditional deployment realized via physical servers running applications, lose attraction in favour to a container deployment era, based on microservices architecture. This shift makes it possible to move towards continuous delivery, maximising deployment speed and flexibility. The Cloud Native Computing Foundation (CNCF) survey published at the end of 2020 showed that 92% of respondents are using containers technologies in production, with a 300% increase over the last four years[2].
These trends bring new security challenges and perspectives. The "castle-and-moat" approach, based on securing the perimeter with a clear distinction between the internal trusted zone and the external untrusted network, is no more applicable in a de-perimetralized era in which there are no defined and physical perimeters and with data spread across devices and multiple platforms. Already in 2004, thanks to the Jericho Forum group[1], there was raising awareness about the difficulties of defining perimeters to secure them. It progressively leads to the necessity of de-perimetralize the security itself, removing the concept of trusted zones, but rather always verifying accesses and actions. These principles have been formalized in 2020 by the National Institute of Standards and Technology (NIST) and National Cyber Security Center of Excellence (NCCoE) inside the NIST special publication 800-207 "Zero Trust Architecture"[2]. Nowadays is becoming a hot topic with governance recommending to migrate or implement Zero Trust network solutions[3].
A Zero Trust network approach leads to gain better visibility of the network and traffic flow, reduce exposition of confidential data, reduce the impact of breach limiting lateral movements and providing each entity with a least privileged model and an overall better security posture and consciousness that security threats must be assumed, both from the internal and external network. From a point of view of technologies, the orchestrator Kubernetes, with 91% of utilization, has become the de-facto standard, providing not only a solid software capable of handling the deployment of microservices, but also an extensive ecosystem in favour of resiliency, modularity and extensibility. These characteristics permit to build on top of Kubernetes other technologies capable of coexisting and enriching the capabilities of the orchestrator. Doing so, it is possible to achieve a *separation of concerns* between Kubernetes, in charge of continuously enforcing the desired state of deployments inside the cluster, the container network interface (CNI) that manages the low-level network connectivity of the cluster, and the service mesh, sitting on the top of the CNI, to handle service-to-service communication providing additional capabilities about traffic

[1] https://www.forbes.com/sites/forbestechcouncil/2021/01/15/how-the-pandemic-has-accelerated-cloud-adoption

[2] https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf

[3] https://www.ncsc.gov.uk/collection/zero-trust-architecture

management, security and observability. The latter technology, adopted in this work with one of the most common solutions named *Istio*, is based on a data plane, made of a distributed set of proxy containers, each one inside each deployed pod, and a control plane capable of centralizing their management, providing configurations and collecting metrics. By doing so, all the traffic will flow via the proxies that will redirect it and enforce policies. This capillary distribution of smart enforcement points leads to several possibilities: the most common and needed network and security features are decoupled from the main application logic making the microservices less prone to bugs and vulnerabilities, and, furthermore, zero trust principles can be injected inside the cluster. It mainly involves the so-called "East-West" traffic of the cluster, composed of the service to service communication that is the foundation of the microservice approach: it is necessary for microservices to work cooperatively in order to provide a service. It implicitly means that interactions between services are numerous and will contain sensitive data that has to be secured. Implementing the Istio service mesh it is possible to enforce encryption, authentication and authorization policies to enable service-to-service authentication, end-user authentication and access control for each workload.

Taking a step further, Istio itself provides extensibility features permitting to apply filters on the traffic and create custom ones. This concept, based on in-proxy extensions made via the sandboxing technology WebAssembly, leads to the possibility of implementing even more granular and flexible custom security controls. Specifically, a Web Application Firewall has been designed and developed based on the open-source Modsecurity library and the OWASP Core Rule Set for the detection of malicious patterns. Its deployment in the network, as well as its configuration, takes place directly at a high level through YAML-based configuration files. Doing so, Istio is exploited to centralize the management of these security controls distributed in a punctual and granular way wherever necessary, at a pod level. Potentially, the deployment of these security controls based on WASM, is not restricted to this use case, but it may be done on any Envoy proxy and, even further, on any proxy that supports the proxy-WASM extensions thanks to a proxy-agnostic development of the APIs.

More in detail, Chapter 2 introduces the current diffusion of cloud solutions, with its service and deployment models and the security issues that derive. Chapter 3 presents the Kubernetes ecosystem with its basic elements and features that may help its security posture, together with some open challenges that will be addressed later on. Chapter 4 summarize the old concept of perimetral security and exposes the Zero Trust principles that should be applied to a network. Chapter 5 introduces the concept of service mesh and analyzes Istio capabilities, leading to how they can be used to implement some Zero Trust principles inside a Kubernetes cluster. Later on, Chapter 6 introduces the possible ways to extend Istio capabilities diving deeper into the WebAssembly technology and its integration with the Istio data plane. Chapter 7 describes at a high level the web application firewall and ModSecurity capabilities. The latter will be applied in Chapter 8 in which the design and implementation of the WASM WAF module are covered. Chapter 9, based on a demo scenario, provides practical examples of Zero trust principles applied with the Istio functionalities and the implemented WAF extension. Finally, in Chapter 10, after testing the latency overhead introduced by adding these functional and security layers to a Kubernetes deployment, scalability, functionality and effectiveness tests are performed against the developed WAF WASM module.

Chapter 2

Cloud Computing paradigm and security implications

This chapter contains a brief description of the state of cloud computing today, the de-facto platform for the deployment of digital services. After some statistics to understand the diffusion and current growth of this paradigm, the concept of microservices is introduced to outline the technological approach. Finally, the chapter ends by introducing the security challenges associated with the rise of the cloud.

2.1 The Cloud today

Cloud computing, conceptually seen as taking advantage of computing resources available somewhere far from us, has become in recent years the most popular paradigm to access and deploy services in a fast, scalable and efficient way. Today already 94% of enterprises use some cloud services[1] and, according to Gartner studies[2], more than 40% of all enterprise workloads will be deployed in public cloud infrastructure and platform services (CIPS) by 2023 with an overall cloud computing market which will be more than doubled in the following five years. Its pervasive diffusion is due to the multiplicity of service and deployment models that can fit most scenarios, and the overall advantages obtained. The latter can be summarized as follow:

- *Elasticity*: Services become on-demand with high flexibility of scaling up and down the resources according to their current needs.
- *Availability*: Services become remotely available from any place and at any time with service level agreements (SLA) that guarantee downtimes extremely limited.
- *Separation of concerns*: Companies can priorities the effort on the core business delegating time and resource-consuming tasks such as the management of infrastructure or system-level details. This results in a recomposition of the business structure outsourcing several IT processes.
- *Economic considerations*: There is a shift from initial costly investments (CAPEX) to an OPEX-based approach: a pay-per-use model permits to pay just the used resources with high flexibility and reduced investments risks.

Regarding service and deployment models, figure 2.1, provided by the NIST, summarizes them all.

[1] https://www.cloudwards.net/cloud-computing-statistics/

[2] https://www.gartner.com/smarterwithgartner/gartner-predicts-the-future-of-cloud-and-edge-infrastructure/

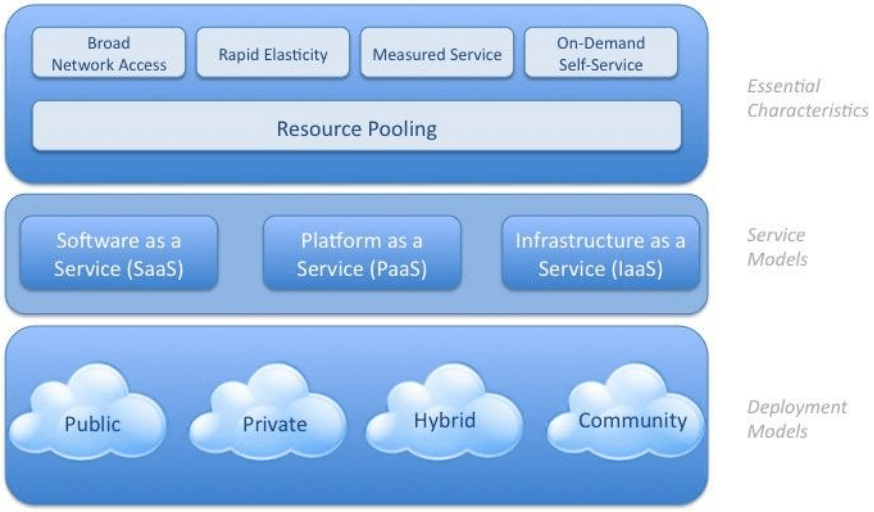

Figure 2.1: NIST Model of Cloud Computing (source: Researchgate)

Depending on the context of application and needs, the service can be provided just as management of the physical utilities (Hardware as a Service model), as a virtualized machine in which full control of the infrastructure is still possible, up to Platform as a Service (Paas) and Software as a Service (SaaS) solutions, suitable respectively for the deployment of developed software on a provided infrastructure and for access of widely used fully functional applications. In addition to this, each service model can be deployed in multiple ways [3]:

- *Private*: Private data center for dedicated hardware or premises solutions. It could be used for several reasons: from the security introduced by using hardware for a single tenant and purpose to the necessity of managing large amounts of data without considerable transfer times.
- *Public*: Pay-to-use format renting virtual servers on multi-tenancies platforms.
- *Hybrid*: A mixed solution, useful to split high confidential services and data to less critical ones, or to manage traffic peaks partially moving the workload to the public cloud when needed.
- *Community*: Different organizations that belong to a specific community share the cloud infrastructure. This is the case, among others, of government initiatives.

Looking at current statistics from CISCO[3], despite various possible combinations of service and deployment, there is widespread use of SaaS solutions (which will reach 75% of total cloud workloads by the end of 2021) and a considerable deployment based on public cloud (Its market share in 2020 was 78%[4]).
The infrastructure itself, as well as the data centers, are constantly evolving to cope with the connectivity needs dictated by a significant increase in traffic compared to on-premises solutions. Their expansion, coupled with the leveraging of innovative technologies such as 5G, allows most applications to migrate to the cloud. This consideration introduces one of the implicit limitations of the cloud: the need for reliable network connectivity. Any application that requires low latency, or full operability even in case of loss of connectivity, is not compatible with service running on remote locations.

[3] https://newsroom.cisco.com/press-release-content?articleId=1908858

[4] https://www.t4.ai/industry/cloud-computing-market-share

2.2 Microservices architecture

From a technological point of view, cloud computing is based on the software architectural pattern of microservices. Pursuing the *single responsibility principle*, applications are no more developed as a monolith app, but rather are decomposed into small pieces that, collaborating, will be capable of providing a service. A single logic unit is named *microservice* and handles a specific business function [4].

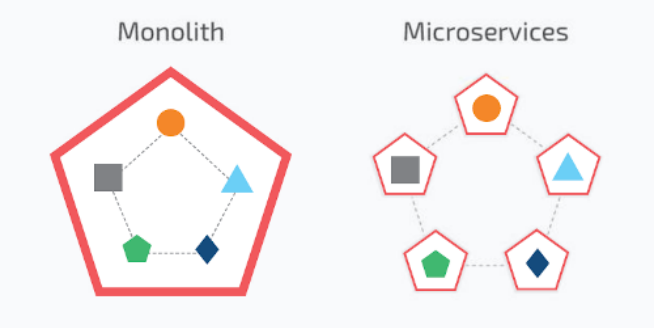

Figure 2.2: monolithic approach vs microservices (source: xme)

This approach provides:

- *Scalability*: Each microservice can be scaled individually and dynamically. Therefore, anymore the entire monolithic application is scaled, but individual services.
- *Improved fault isolation*: If a specific microservice breaks, the overall application will still provide all the others functionalities. Furthermore, each microservice can have more instances, even on multiple machines, leading to a higher probability to have one of them fully operative.
- *Development agility*: Running multiple instances of the same microservices leads to the capability of easily test and switch between different versions of the software or upgrading it in a scalable way.
- *Development flexibility*: Each microservice can be developed individually with total freedom in terms of choosing the appropriate technology stack (e.g. programming language used).

From the point of view of the challenges introduced, they can be summarized in the following list:

- *Development*: To fully leverage the advantages introduced by the cloud, the applications have to be redesigned following cloud-native rules[5]. Moreover, additional complexity is present due to the creation of a distributed system.
- *Debugging*: Specific approaches and techniques are needed to debug and test distributed systems, resulting in complex tasks.
- *Observability and Tracing*: Complexity is introduced also because the overall distributed infrastructure becomes more challenging to understand. Each microservice runs independently and asynchronously with difficulty on tracking the user's requests and reproducing bugs.
- *Security*: The interaction between the microservices in order to produce the response to the user takes place through the network, typically with application protocols such as HTTP and gRPC. A security cloud perspective based on network security is needed. The following section (2.3) provides a full overview about it.

[5] https://12factor.net/

Microservices, and their underlying containers, require structured software capable of automating deployment, scaling and managing containerized applications. Nowadays, as stated by the 2021 Sysdig Container Security And Usage Report[6], the de-facto standard orchestrator is the open-source platform *Kubernetes*. Figure 2.3 shows a 75% of Kubernetes usage that reaches up to 91% considering all the orchestrators based on Kubernetes itself.

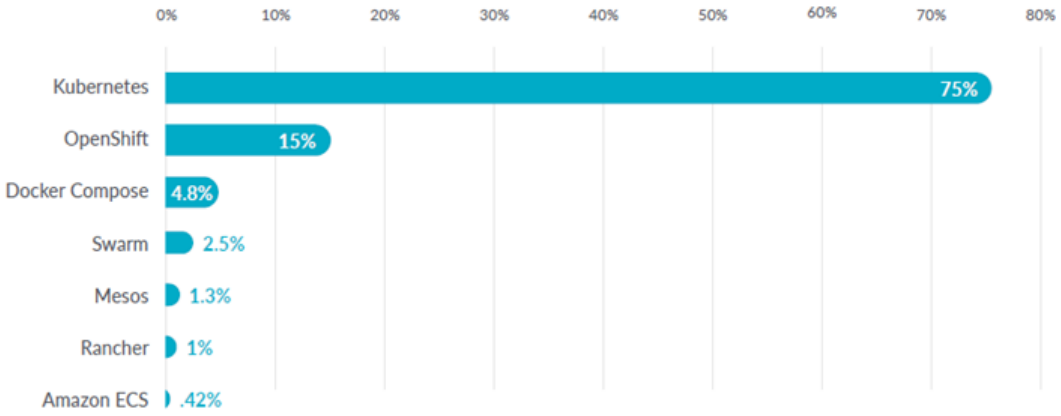

Figure 2.3: Container orchestration platforms usage (source: Sysdig)

2.3 Cloud security

The aforehand mentioned movement of users and enterprises into the cloud leads to the necessity of taking into consideration security concerns. This is highlighted even more by statistics about cyber attacks. The 2021 Thales Data Threat Report[7], based on a survey with more than 2600 security professionals and 950 companies worldwide, reports that only 20% of them was sufficiently prepared on handle the new business operating environment considerably enhanced by the Covid-19 pandemic. It resulted in more than 41% of them that experienced a security breach in the last twelve months.
The traditional perimeter network defense, based on a distinct separation between an internal secure network and the untrusted one outside, has become insufficient. Nowadays users and employees expects to be able to access their data and have full capabilities anytime and anywhere in terms of physical location and devices. It means that a physical approach working on the on-premises network to delimit what is trusted or not is obsolete[8]. To further understand the reasons, below are listed assumptions no longer valid in the cloud era:

- *Frontal Attacks*: Perimeter is no more defined leading to have a surface of attack way wider. Sensitive data are not confined and attacks may target plenty of different entry points. A traditional firewall deployment becomes not enough to properly handle them all.
- *Trusted inside*: This assumption leads to uncontrolled possibility of lateral movements once the perimeter is breached. There should not be a distinct delimitation between trusted and untrusted origins. By default the security design should follow the *least privilege principle*: any type of entity like individuals or processes, must have the minimum privileges and resources necessary to complete their tasks[9]. Even further, in conjunction with strong authentication mechanisms, internal actions should be monitored to be aware of unauthorized

[6] https://dig.sysdig.com/c/pf-2021-container-security-and-usage-report

[7] https://cpl.thalesgroup.com/data-threat-report

[8] https://www.microsoft.com/security/blog/2019/10/23/perimeter-based-network-defense-transform-zero-trust-model/

[9] https://csrc.nist.gov/glossary/term/principle_of_least_privilege

access and actions performed via compromised credentials. 2021 Data Breach Investigations Report[10] provided by Verizon highlights that 61% of breaches involved credentials compared to only a 3% that exploited a software vulnerability.

Cloud Computing introduces also consideration about who is the entity in charge of guarantee security solutions. It is the so-called *Cloud responsibility* between cloud providers and customers. Providers have to guarantee specific security standards defined via SLA, but security has to be a shared responsibility even with the costumers with differences based on the deployment model adopted [5]. IaaS requires fulfilling the role of the system administrator who manages the entire logical architecture: starting from the firewall to security updates on the operative system (OS) and applications. PaaS requires taking care of application-level vulnerabilities, and even Saas, the model that delegates the most to the provider, still requires apply identity and access management (IAM).
In addition, even the deployment model requires some considerations: a private cloud solution will intrinsically have lower security risks rather than a public cloud in which arise problems related to sharing physical hardware with other tenants, the consciousness of the actual resource location and ownership. The complexity increases even more taking into account the *multi-cloud*: in this case services, and data, are split across more than one cloud service provider.

2.3.1 Cloud Security Issues

To provide an easy to catch overview of all the cloud security issues, figure 2.4 gives a categorization of them. *Security policies* includes all the SLAs agreed and standard that must be applied to obtain a compliant system. Nevertheless, compliance-driven security does not have to be the goal, but rather a first step of the bigger picture of security. *User-oriented security* issues are related about the fact that, given the complexity of the cloud, an user centric approach must be adopted via authentication, authorization and IAM. *Data storage* issues shows the necessity of secure the data, both in transfer and in storage taking into account confidentiality, integrity, and availability (CIA triad) [6]. *Applications* vulnerabilities remains one of the top concerns: the multiplicity of programming languages, libraries and frameworks, joined with the shift to a cloud native approach, requires skilled developers aware of security issues. It is not restricted to application development, but also to OS development: the OS itself, with the relative kernel, plays a major role guaranteeing lightweight virtualization primitives with process isolation capabilities. *Network* plays a major role with the cloud computing highly dependent on it. Data must come across the network in a safe way, but it is also needed to have mechanisms capable of filtering, monitoring, logging and preventing malicious access.

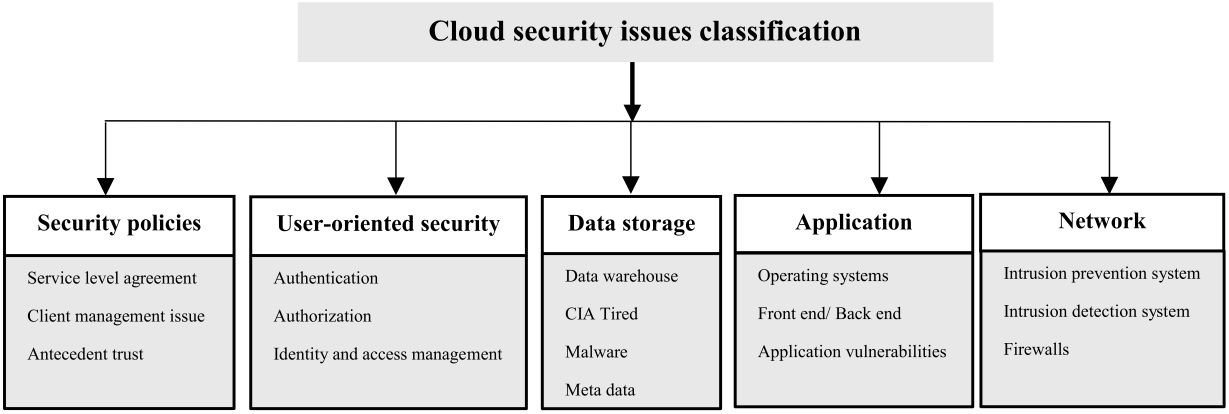

Figure 2.4: Cloud security issues classification (source: [5])

[10] https://enterprise.verizon.com/resources/reports/2021/2021-data-breach-investigations-report.pdf

There are several papers in the literature that present these issues in depth [5, 6, 7]. This thesis, as introduced, rather aims at showing and proposing the implementation of a new architectural security model approach that is more effectively adaptable to the complexity of the modern environment introduced by the well-established cloud computing. By doing so, it will be possible to effectively deal with several cloud issues mentioned above.

Chapter 3

Kubernetes orchestration: security capabilities and open challenges

The rise of cloud and the relative shift to microservices required a new approach in terms of deployment operations (several applications must be handled instead of the single monolithic one), scaling and management of the lifecycle of the container and, consequently, of the relative running applications. All these key requirements led to the necessity of a robust and innovative software capable of creating a higher level of abstraction: removing the complexity of deployment operations and implicitly making the whole system flexible, resilient and efficient. For these purposes Kubernetes has been created. Today, as stated in 2.2, it is the de-facto standard container orchestrator platform. It was initially developed by Google and since 2016 it has been managed by the Cloud Native Computing Foundation (CNCF) with a huge contribution from the open source community.
This chapter aims to provide, after a brief introduction of the features, the basic concepts about the Kubernetes architecture and its components, useful for a correct understanding of the following work. The focus will not be about the entirety of capabilities that a complex tool like Kubernetes is capable to provide, but rather on elements with a correlation with its security posture. For further details, there will be references to the extensive documentation provided by the Kubernetes project itself [8]. The chapter ends exposing the native security features available and the open challenges, in terms of networking and security, that the thesis aims to take into account.

3.1 Features

The core features of Kubernetes can be listed as follows:

- *Scheduling*: The capability of automatically figure out the most suitable location in which a container must be executed. It can be decided according to several parameters like the actual workload of each machine, minimal and suggested resourced required by the container, affinity requirements with other containers and labels to keep possible to manually assign containers to a specific machine (e.g. testing machine).
- *Replication and load balancing*: Via replication is it possible to create multiple instances of the same container. It includes the capability of properly manage the workload across all the replicas thanks to load balancing capabilities.
- *Horizontal auto-scaling*: The creation of replica can be escalated to dynamically handle the incoming workload that may have great variations. Scaling up and down the replicas permits to handle spikes in a efficient way in terms of resources and, consequently, costs.

- *Service discovery*: Containers and relative replicas lifecycles become really dynamic, to manage it Kubernetes provides and efficient way to find the correct container to interact with and so handling in a reliable way the communication between the microservices.

- *Self-healing*: To guarantee high availability inside a system that manages hundreds of containers a functionality capable of handle errors is needed. Kubernetes, checking continuously the health of each containers provides a self-healing feature that automatically instantiate a new replica in case of ant fatal error. This has an impact also in terms of machine failures. Managing multiple machines the new replica can be assigned executed to another running one reducing down-times and service disruptions.

- *Automated rollouts and rollbacks*: Changes are propagated to all the container interested in, updating the application deployed or its configuration. Keeping an history of previous rollouts, Kubernetes provides also a fast way to recover from errors allowing to perform rollbacks to previous states of the deployment.

- *Secret and configuration management*: Kubernetes provides features to decuple configuration elements and sensitive ones like tokens, keys and credentials outside the containers. This leads to flexibility on updating them without the necessity of rebuilding the image and a reduced exposure of sensitive data.

- *Provisioning Storage*: Given the fact that files stored inside a running container are ephemeral, the necessity of storage becomes clear for any stateful application like databases. To do so, specific functionalities capable of managing the storage are provided. It is possible to link to a container defined persistent volumes that can be local, managed by an independent network storage system or hosted somewhere else, for example on a public cloud.

All these core features, combined with a high level of flexibility both in terms of configurability, extensibility of the orchestrator itself and in terms of deployment models (Kubernetes can run on any type of infrastructure), have determined its great popularity compared to alternatives such as Docker swarm, which is restricted to docker containers orchestration and has a smaller number of functionalities. As the next chapters will present, extensibility and flexibility have been key elements in this work, allowing additional layers of functionality to be added on top of a solid tool like Kubernetes.

3.2 Architecture

The Kubernetes architecture is based on a master node and one or more workers. The master node is the control plane with all the components capable of managing the cluster. All the user's container, instead, are scheduled by the master node to a worker node and are going to be executed there. Talking about Kubernetes, a fundamental clarification must be made: the smallest unit that can be scheduled is called a *pod*, which can have one or more containers inside. The master mode will therefore schedule pods.

3.2.1 Master node

The master node is the main logic: manages the scheduling, reacts to events modifying the status of the cluster and monitors the overall activity. To guarantee availability it can also have multiple instances across different physical machines. As shown in figure 3.1 it is composed by several components:

- *API Server*: It is the entry point to control and change the status of the cluster. Any interaction starts contacting the API server, both from users and programs, scripts or any other component.

- *Etcd*: It is a distributed key/value store that keeps the state of all the elements of the cluster. It provides cluster persistence. Keeping in memory the desired status, as soon as it changes, watchers are triggered to actually performs all the required steps. This is one of the main functionalities of the master node: to constantly monitor the actual state keeping or adapting it based on the desired state.
- *Scheduler*: It is the responsible to schedule the pods, in which the applications run, across the cluster selecting the most suitable worker node.
- *Controller manager*: Implements the control loop: an endless loop that keeps listening to controllers and reacts accordingly sending messages to the API server. Controllers are watchers of a Kubernetes resource, or state of the cluster and are triggered if the current state differs from the desired one. For example, controllers can monitor if a worker node fails or monitoring the number of instances of a specific container.

The cloud Controller manager showed in figure 3.1 is present only in case of the presence of a cloud provider. It is a layer to manage specific controllers that interact with the cloud provider APIs to permit interaction between the Kubernetes cluster and the cloud platform and keeping Kubernetes flexible and independent. For the purposes of this thesis, this component will not be used.

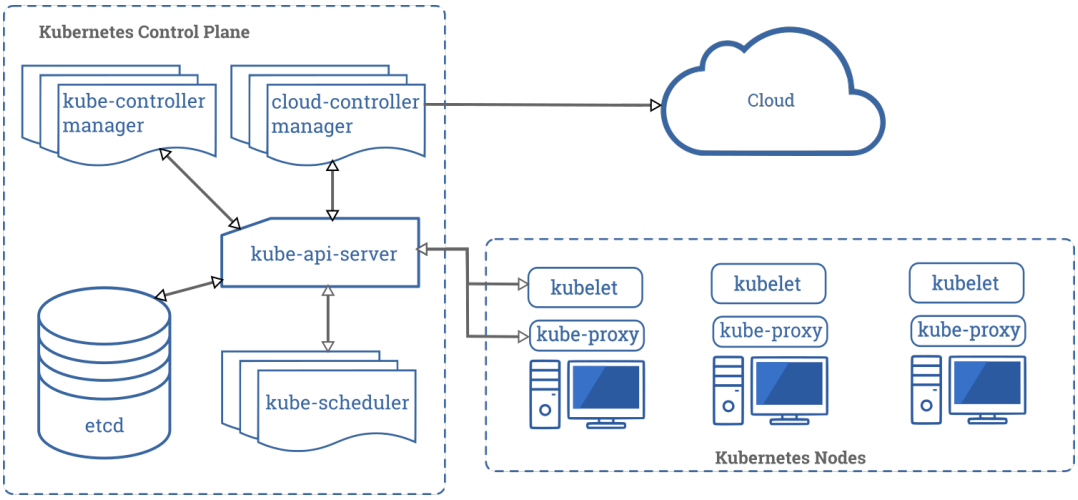

Figure 3.1: Main components of Kubernetes (source: Kubernetes.io)

3.2.2 Worker node

User's workloads are executed inside the worker nodes. The control plane sends requests to be accomplished and the designated nodes process them. The key components, present in each worker node (see figure 3.1), are:

- *Kubelet*: It is responsible to run the pods, to keep under control their status, to update about them the control plane and to interact with the container engine to properly perform these functionalities.
- *Container engine*: The actual execution of the container is delegated to the container engine. Kubernetes supports several of them: initially the default one was Docker, now deprecated for optimization reasons in favour of *containerd* or any other container engine compatible with the Container Runtime Interface such as CRI-O[1].

[1] https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/

- *Kube-proxy*: It is one of the key element of kubernetes networking: a network proxy to maintain and apply the network rules defined in the cluster.

Note that these components are also present in the master node. It will also run pods to provide its system functionalities.

3.3 Main resources

Kubernetes provides a set of resources made to work together, usually one on the top of another. They can also be seen as a pillar of abstractions: starting from the actual application running inside a container up to a high-level resource in which there is the description of which and how the application must be deployed. Using these resources it is possible to describe the desired status of the cluster and manage the lifecycle of all the applications running on it. In Kubernetes terms, an application takes the name of *workload*. A workload runs inside one or multiple pods.

3.3.1 Pod

As mentioned above, a pod is the smallest deployable unit and wraps one or more containers. This level of abstraction, compared to directly deploy containers, permits to have the core application inside a container and to keep, alongside it, any other capability logically related. Secondary containers that run along with the main one take the name of *sidecar containers* and are usually used to perform helper tasks like data pullers, data pushers, and network-related tasks such as proxies. All the containers inside a pod will share the network namespace: resulting in having a common pod's IP address and the possibility to communicate with each other through the loopback network interface. Pods, by their nature, are ephemeral entities: they do not store data that must be persisted and, if errors lead to fails of the pod or its relative node, the pod will not be healed or rescheduled, but rather a new replica of it will be spawned.

3.3.2 ReplicaSet

The horizontal scalability of Kubernetes is based on replicating the pods to properly handle all the requests. ReplicaSets are the resources that, based on a pod template, instantiate the pods and maintain stable their number. It means that a ReplicaSet will be in charge of initially generate the right number of pods, replacing pods that generated errors with new ones and terminating some of them based on the details defined inside the deployment.

3.3.3 Deployment

It is recommended to do not manually create ReplicaSets, but rather delegate their creation and management to another resource: the deployment. This resource wraps in one place all the information needed to deploy a workload: the details about the pod, the number of replicas wanted and features for updating the pods, performing rollbacks, dynamically scaling the workload and so on[2]. Just like any other resource, a deployment can be expressed via a *YAML file*. The Kubernetes command-line tool (kubectl), typically used to provide these files to Kubernetes, will be in charge of converting the information into API requests. A YAML file is a configuration file in which maps with key-value pairs and lists of an indefinite number of elements can be written and nested. It is preferred to JSON because it is more user-friendly and has better readability. Figure 3.2 shows a YAML file example used to describe a basic deployment. In a human-readable way, this file describes a deployment of a workload based on a specific container image tagged as `nginx:1.20` and with a total of three replicas that will be handled by the ReplicaSet.

[2] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: nginx-deployment
   labels:
      app: nginx
spec:
   replicas: 3
   selector:
      matchLabels:
         app: nginx
   template:
      metadata:
         labels:
            app: nginx
      spec:
         containers:
         - name: nginx
           image: nginx:1.20
           ports:
           - containerPort: 80
```

Figure 3.2: Basic nginx deployment described via YAML file

3.3.4 Service

Deployments, with replicaSets and pods, permit to make up and running the workload, but still, the service resource is needed to effectively interact with it. This need arises from the fact that pods are ephemeral, it is not possible to use their IP address reliably. The service resource, defined as *an abstraction which defines a logical set of Pods*, removes all the complexity of managing the dynamic nature of pods and provides a stable IP address and a DNS name to interact with the workload. The service can be exposed in three different ways:

1. *ClusterIP*: The default option, the IP address provided is accessible from inside the cluster.
2. *NodePort*: Selecting a port number (from a range between 30000-32767), the service becomes accessible from outside the cluster. The IP address will be the one of the node. Internally a ClusterIP is also created and the request will be routed to it.
3. *Load Balancer*: The service is externally exposed with a dedicated IP address.

The binding between a service and the pods that handle is made by *labels*, key-value pairs strings that go hand-in-hand with a given object. All the pods with the label defined inside the `selector` field of the service YAML declaration will be dynamically identified by the service that will route the request to them. In the case of replicas, the round-robin algorithm is used to select a pod. Figure 3.3 summarize in a graphical way the idea behind pods, replicas and services. Service A is the entry point to a workload labelled as `app:WorkloadA`, with a single pod. The pod, following its logic, has to contact service B, which will provide the request to one of the replicas labelled by `app:WorkloadB`. Internally each pod will have at least one container.

3.3.5 Persistent-Volume

The purpose of persistent volumes is to provide storage for a pod on the local file system. Via another resource called *persistent-volume claim*, the application makes an access request to the persistent volume. This solution to implement persistence is still not suitable in many cases: local

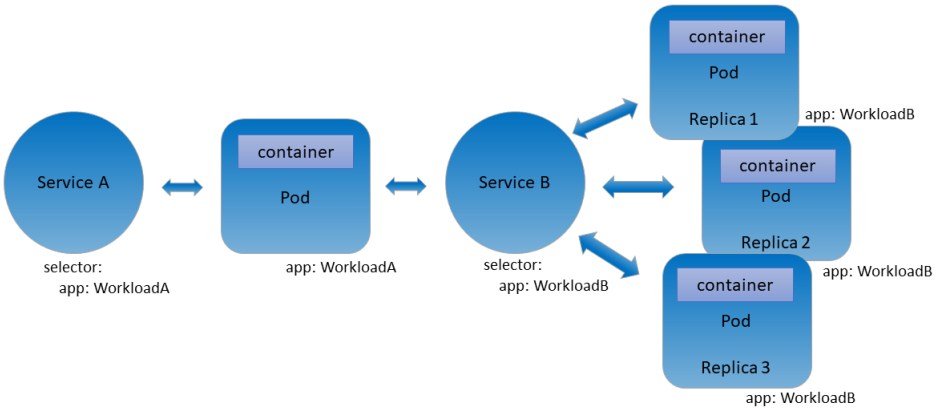

Figure 3.3: Interaction between Services and Pods

storage binds to application to a specific node making it more difficult to schedule and reducing the resilience of that workload. More suitable solutions are based on tools like NFS (Network File System) that provide dynamic allocation.

3.3.6 Namespace

An important Kubernetes feature to point out are the namespaces. Through namespaces, it is possible to partition the cluster into smaller virtual clusters in which resources can be organized and aggregated. This becomes useful to manage complex applications, avoid conflicts between teams that work on the same cluster and it has also security implications: it is indeed possible to limit the access to specific namespaces, define resource quotas to limit the resources that each namespace may consume and apply policies namespace-wide like, for example, network policies. Further security considertions about namespaces are presented in 3.4.3. By default Kubernetes already uses namespaces to split he system resources (`Kube-system` namespace) from the user's resources (`Default`) and the cluster public information data (`Kube-public`). It is possible to create namespaces freely, a cluster commonly has more of them aggregating functionalities such as workloads for monitoring or other features added to the cluster. Most of the resources can be assigned to a specific namespace. The usual way is by specifying the namespace name inside the metadata field with the key `namespace` of the YAML file. These resources will not be accessible from another namespace. Other low-level resources, on the other hand, are cluster-wide. Examples of the latter are persistent volumes and namespaces themselves (namespaces nesting is not permitted).

3.3.7 Custom Resource Definition

Kubernetes Resource model leads to a common structure for all the resources. It enables the possibility to create custom ones. The overall outcome is a high capability of extending the functionality of the cluster. Once defined, the custom resources are fully integrated inside Kubernetes, permitting the user to use the new resource just like any built-in resource and to persist it inside etcd, the cluster database. Going further, the *operator pattern* combines custom resources and *custom controllers*. Based on the information provided by the resource, it will be in charge of automating the process of enforcing the specifications applying changes on the cluster. This high degree of modularity provides the basis for considering Kubernetes not just a container orchestrator, but rather an ecosystem capable of extending itself with new, fully integrated, features.

3.3.8 Container Network Interface

Following the modularity principle of Kubernetes, the networking is not managed by Kubernetes itself, but rather the orchestrator will just be a consumer of network functionalities provided by a

network plugin: the *Container network interface* (CNI). The CNI plugin is in charge of creating the virtual interface of the pod, assigning the IP, connectivity features and any other specifications to configure the networking. The default one takes the name of *kubelet*, but several alternatives may be used for more advanced features or specific purposes. Common alternatives are Flannel, Weave, Calico and Cilium. The latter, given its performance and safety features, will be discussed in more detail.

3.4 Security features

3.4.1 Securing the API Server

Kubernetes' security capabilities are mainly focused on securing the API Server, the entry point to control the cluster. The first defence, therefore, is to control the access and action performed inside the cluster. The entities that may want to interact with the server API are physical users via the Kubernetes command-line tool (kubectl), client libraries or REST requests and pods. For the latter, a specific resource called *service account* is linked to each of them providing an identity. First of all, Kubernetes requires that all the communications to the API server are encrypted.

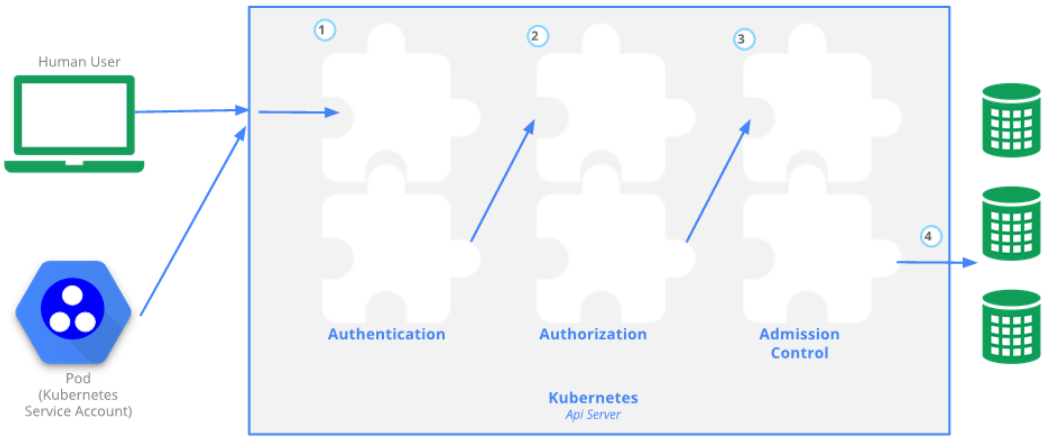

Figure 3.4: Security steps on accessing the Kubernetes API (source: Kubernetes.io)

Kubernetes itself is capable of acting as Certificate Authority (CA) signing and distributing certificates to all the components that will interact with the API server. Doing this, the API server will serve only TLS connections on port 443. As shown in figure 3.4, regardless of the actor of the action, whether it is a user account or a service account, each encrypted request that reaches the API server has to pass through three phases before being processed:

1. *Authentication*: Different strategies are possible. For user accounts, there is not a corresponding Kubernetes resource like a service account, but rather a user may be defined based on its client certificate signature: any certificate signed by the cluster's CA will be considered valid and the user authenticated. Alternatives are based on different types of tokens: static ones defined inside a static token file of the cluster or dynamically managed ones, called *bootstrap token*, stored as secret resources inside the cluster. These tokens have to be sent attached to the request inside the HTTP Authorization header. Service accounts authentication is also based on tokens. In this case, they are automatically mounted inside the pods.

2. *Authorization*: The authenticated request is authorized if there is a policy that allows the user to perform the requested action. There are several authorization modes (ABAC, Node, Webhook etc.), but the suggested one because it is more intuitive and easier to manage is the *Role-based access control* (RBAC). Via RBAC is it possible to define *Role* (to limit permissions to a specific namespace) and *ClusterRole* (for permissions to the entire cluster).

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
spec:
  podSelector:
    matchLabels:
      app: workloadA
  policyTypes:
  - Ingress
ingress: []
```

Figure 3.5: NetworkPolicy example, deny all

Through *RoleBinding* and *ClusterRoleBinding* they are then associated with users. One important thing to point out is that permissions are just additive, with Kubernetes it is not possible to explicit a deny rule.

3. *Admission control*: This is the last step that the request must go through. Admission control works just on requests that write/edit objects, it is a sequence of modules that can reject the edit or customize it (e.g adding default fields). By default, Kubernetes provides a long list of Admission controls that can be enabled[3].

Only at the end of the admission control phase, the request is processed applying all the changes to etcd and consequently triggering all the actions on the whole cluster. A protected Kubernetes API is certainly a key factor for a secure cluster, but, analyzing the overall picture with a defense in depth approach, it is just one of the layers that should be secured. It is interesting to also take into account the interaction between the workloads and their security posture. The point of view, therefore, becomes more about networking and the attack surface provided by the applications running inside the cluster.

3.4.2 Network policies

By default, pods access is unrestricted. It means that inside the cluster they accept traffic from any origin. Network policies, working on IP address or port level (OSI layer 3 or 4), permit to isolate them: as soon as a network policy is activated specifying a workload's label, all the relative pods become isolated. This happens because network policies work in a whitelisting way, allowing only the traffic permitted by the policy. This feature becomes extremely important to reduce network exposure taking into account the likelihood of compromised pods and lateral movement from them. Having the unrestricted possibility to interact with all the user's pods of the cluster may lead to serious security concerns such as leakage of credentials and sensitive information, privilege escalation and malicious use of the cluster resources (e.g. cryptocurrency mining).
Figure 3.5 shows an example of NetworkPolicy definition. The key `name` inside `metadata` provides a unique policy name, via `podSelector` the label of the pods on which the policy has to be applied is specified and `policyTypes` lists the type of rules that may be applied (for ingress or egress traffic) and all the rules for each type of traffic. In this example, no further elements are listed after `- Ingress` (`ingess` is an empy array), therefore no rule authorising incoming traffic is applied. Given this reason, as the name states, this rule is called *deny-all*. Although this rule may appear elementary, it is an important start to make applications inside the cluster more secure: it should be used as the first rule to start whitelisting only specific incoming traffic or to

[3] https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

```
ingress:
  - from:
    - podSelector:
        matchLabels:
          app: workloadB
```

Figure 3.6: NetworkPolicy, snippet of allow rule

temporarily isolate one pod for further investigation. Figure 3.6, shows a snippet of code that can complete figure 3.5 providing one ingress rule. In this case, all the pods labeled with `workloadB` will be able to contact `workloadA`. More detailed policies can be written by combining labels of pods, namespaces and IP blocks allowed.
Note that the implementation of network policies, just like most of the other network-related aspects, is delegated to the CNI (see section 3.3.8). It is therefore a prerequisite that the CNI used supports this functionality. In some cases, for example using *Calico*[4], the CNI itself also provides specific implementations of network policies (made via CRD) with custom features.

3.4.3 Namespaces: security capabilities

About namespaces, further consideration can be done. As introduced in section 3.3.6, logically partitioning the cluster into virtual ones, leads to security benefits. However, it is important to note that this logical division may lead to false assumptions. In anticipation of subsequent considerations, the enforcement of security inside a Kubernetes cluster is mostly left to the system administrator. Security features related to namespaces are not an exception. The division into namespaces makes it possible to aggregate workloads and resources on which to apply, for example, network policies, but this means that, by default, there is not network-level isolation between namespaces. Two pods from different namespaces, knowing the mutual IP address, can freely interact with each other. This can be easily verified by performing a small experiment.
First, create two empty namespaces called `ns-a` and `ns-b`:

```
kubectl create namespace ns-a
kubectl create namespace ns-b
```

Then, deploy a basic nginx pod (see figure 3.2) via kubectl specifying a namespace with the `-n` parameter.

```
kubectl apply -f nginx.yaml -n ns-a
```

Getting the pod's name and describing it, the pod's IP is retrieved.

```
kubectl get pods -n ns-a
kubectl describe pod nginx-deployment-6897679c4b-rgndv -n ns-a
```

On the other namespace, performing the following command, another pod is deployed and an interactive shell is executed to perform curl commands from the pod.

```
kubectl run curl-pod --image=radial/busyboxplus:curl -i --tty --rm -n ns-b
```

Executing `curl 10.42.0.39:80 -v` the response code is `HTTP/1.1 200 OK`, showing the default nginx page.
Having said that, namespaces are certainly a feature that helps to obtain a first level of segmentation of the network, but they must be adequately configured. Doing so and restricting network access, leads to a reduction of the exposed resources and the number of manoeuvres that can

[4] https://www.tigera.io/project-calico/

potentially be performed[9]. Furthermore, resource quotas can be defined for each namespace. It comes in handy limiting the resources that even a compromised namespace can use. It will both help to reduce the impact on other applications running on the cluster and providing fewer resources for malicious purposes.

3.5 Open challenges

Kubernetes provides a complete and flexible ecosystem, but its adoption and widespread usage still includes challenges. A survey conducted by the CNCF (see figure 3.7) shows that *security* and *networking* are the top challenges for Kubernetes users. The following sections will show

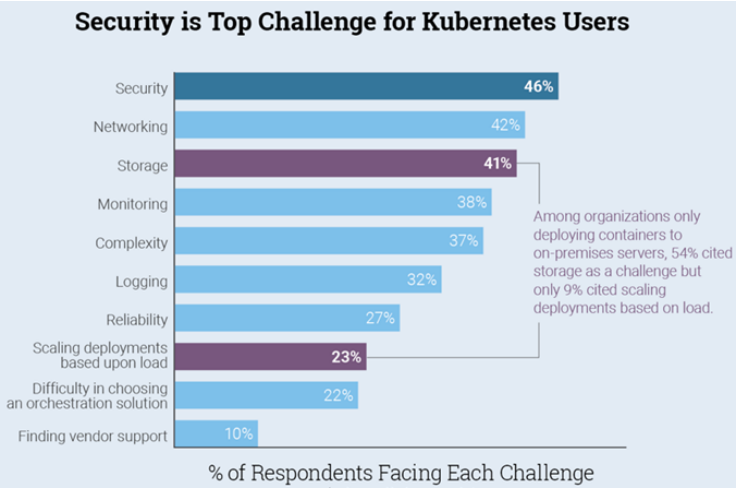

Figure 3.7: Kubernetes top challenges (source: thenewstack)

some of the challenges related to these topics. These challenges are going to be addressed in the next chapters discussing possible solutions.

3.5.1 Networking

- *Third-party libraries for common features*: Useful features for developers are not implemented natively, so extra libraries are needed to be used inside the application. These functionalities, such as circuit breaking, should be delegated to the network level without having to integrate network-aware logic inside the application code.
- *Lack of granularity and basic load-balancing*: Kube-proxy settings lack granularity. There is no possibility of fine-tuning traffic to distribute it between, for instance, different versions of a service. Kubernetes load-balancing, as stated in section 3.3.4, is based on the round-robin algorithm between the healthy replicas. It is a valid way to provide basic load-balancing features between the replicas of a service, but there are not practical (or any) ways to define rules on weight or traffic type that would allow to perform a more advanced load balancing and features like canary deployment.

3.5.2 Security

- *NetworkPolicies*: From a security point of view, network policies certainly represent an important feature (see section 5.6), but still present some limitations. First of all, they are enforced in kernel-space at OSI layer 4. This results in policies that can be based only on IPs and ports apart from the aforementioned Kubernetes resource attributes. There is no way to exploit any application layer attributes such as HTTP request methods and header fields. Another limitation is in terms of semantic due to the fact that network policies are just expressed via allowing rules, not implementing any denying ones.

- *Microservices with more than just the main logic*: The necessity of implementing common network and security features inside the application itself results in: more efforts from developers, a bigger application code more prone to bugs, repetition of similar implementations on different microservices. Not restricting microservices just to their minimal core logic is an overall increase of risk due to an extended attack surface and difficulties to properly manage updates and configurations.
- *Service-to-service plain traffic*: While the interactions with the API server of the control plane network can be encrypted, for data plane connections Kubernetes does not provide any native solutions. This results in the necessity of delegating encryption to 3rd-party software or implementing it directly inside the application, raising the concerns mentioned above. Not addressing this issue would mean trusting the cluster's internal network. At a fist glance it may appear a reasonable risk, but it creates a breeding ground for lateral movements and exposes all the transmitted data to eavesdropping attacks and compromisation. Further considerations will be discussed introducing the zero-trust principles (Chapter 4).
- *Lack of observation and tracing systems*: As introduced in section 2.2, the microservices approach comes with the necessity of adapting monitoring strategies to the new architecture and an overall new set of observability, tracking and debugging challenges. Natively, Kubernetes itself lacks a centralised system to converge and analyse logs and tracing systems to understand the flow of requests. These features are delegated to third-party add-ons such as Prometheus and Jaeger[5].

[5]Overcoming the Challenges of Centralizing Container and Kubernetes Operations, https://www.usenix.org/conference/lisa18/presentation/chunikhin

Chapter 4

Zero Trust security

The Zero Trust (ZT) terminology has been coined by Forrester already at the beginning of the past decade[1], but even earlier, the concept behind it derives from Jericho Forum in 2004 in which the focus was about the de-perimiterization problem and a way to solve it via the necessity of trust[2]. This concept has evolved becoming more extensive and comprehensive and leading to a NIST special publication at the end of 2020 called "Zero Trust Architecture" [2]. This document provides an abstract definition of Zero Trust Architecture (ZTA) and gives general deployment models to embrace ZT. The zero trust paradigm, as will be discussed, represents a new way to approach network security, overcoming the obsolete perimeter security and becoming a key element to inject security principles inside the distributed, dynamic and ephemeral cloud computing world based on a microservices architecture. The latter breaks the traditional architectural schemes requiring, also from the security point of view, a new approach.

4.1 Castle and moat: the perimeter security approach

First of all, before introducing new methodologies, it worth briefly introduce the previous approach. *Perimeter-based network security* is based on a clear distinction between trusted and untrusted entities. This is based on a perimetral distinction between entities inside the network perceived as trusted, not compromised and capable of acting in a responsible way, and the external ones, composed of untrusted entities that may try to access and interact. This approach is also known as *caste and moat*, in which everyone inside the castle is safe thanks to a high perimetral security and a defined way to access. This traditional defense is made by architecturally design the network around firewalls, intrusion detection systems (IDS) and intrusion prevention systems (IPS) deployment. The firewalls must be the only contact point between internal and external networks and only authorized traffic (both incoming and outgoing) will be permitted. This leads to strictly define a perimeter capable of confining outside the external threats. Although this approach certainly includes valuable security controls to secure the network, it was based on a typical static and on-site infrastructure capable of defining also a physical perimeter. The key point to be noted is that this approach, considering infrastructures and data spread across devices and multiple cloud providers and the dynamicity of users, is becoming progressively less viable. Furthermore, implicitly trusting internal users and communications leads to intrinsic security concerns: internal threats, disgruntled or careless users and employees, compromised identities, access to unprotected sensitive data and lateral movements should be aspects to take into account. The concept of zero trust, that will be formally defined in the next section (4.2) reverses this paradigm, there is no longer an implicit trust of any entity, but an implicit awareness that

[1] https://go.forrester.com/blogs/a-look-back-at-zero-trust-never-trust-always-verify/

[2] https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2020/pdf/ADD-1001-Zero-Trust-Security-Model.pdf

everything may be compromised regardless of it is internal or external to the network. NIST illustration (figure 4.1) graphically shows the differences between the two approaches, and how zero trust can complement the traditional firewalled network. It may still be possible to put firewalls ahead of networks, just like in front of the Kubernetes ingress of a cluster, to reduce exposure to the internet, but afterwards the netwok is still not trusted and any implicit trust zone is much smaller.

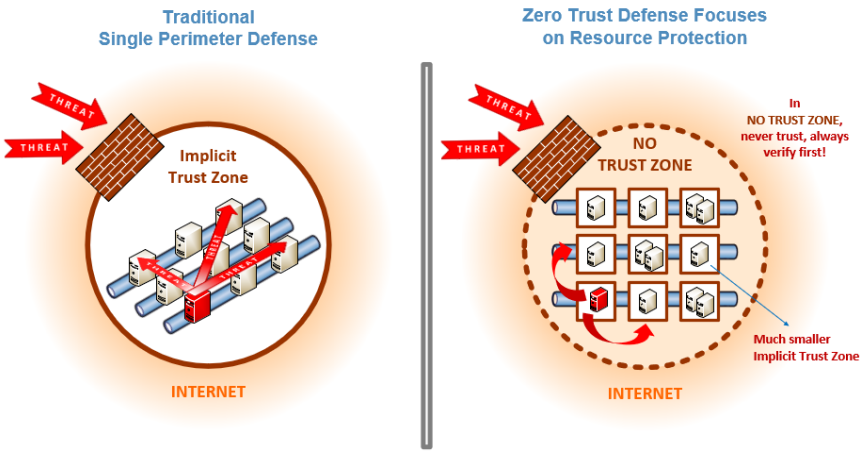

Figure 4.1: Illustration of traditional network vs zero trust network (source: NIST)

4.2 Zero Trust definition and principles

According to NIST definitions [2], *"Zero Trust is a cybersecurity paradigm focused on resource protection and the premise that trust is never granted implicitly but must be continually evaluated"* and *"provides a collection of concepts and ideas designed to minimize uncertainty in enforcing accurate, least privilege per-request access decisions in information systems and services in the face of a network viewed as compromised"*. The first assumption behind this concept is that everything is not safe, there are not secure origins, but rather everything must be verified.

Performing the so-called *micro segmentation of security*, security becomes no more static and strictly defined by a perimeter in which everything inside can be assumed as secure, but it is applied in a more granular way into smaller protected zones that can even converge in just single applications. With this shift in point of view, it becomes clear that zero trust focus is about the protection of resources like assets, workflows, accounts and so on, not network segments. From these concepts, ZTA has been defined as *"an enterprise's cybersecurity plan that utilizes zero trust concepts and encompasses component relationships, workflow planning, and access policies"*.
At a high level, the basic principles for grasping the ZT mindset are:

1. *Secure access for all the resources*: Authentication and authorization must always be performed identifying explicitly and in a secure way the user/device that is requesting a resource. As stated by NIST, there is a shift of the policy decision and enforcement closer to the requested resource.

2. *Least privileged model*: As mentioned in section 2.3, actions must be limited to the necessary ones without exposing any unnecessary resources, but also providing everything that is needed to maintain availability and to do not compromise productivity. In other terms, users access must be *Just In Time* (JIT) and *Just Enough Access* (JEA).

3. *Breach assumption*: The assumption that a breach is inevitable and that threats are already inside the organization, leads to better proactivity and consciousness. The mindset is no more about protecting something to avoid any sort of damage, but rather to mitigate it in any possible scenario, in order to avoid lateral movements that may lead to compromise other sensitive data and workloads.

A more verbose description, with an enterprise-centric perspective, can be found inside the NIST document [2] at section *2.1: Tenets of Zero Trust*. For the purposes of this thesis, in which these concepts are going to be applied on a cloud domain based on microservices, the focus will be mainly on the network. It is, therefore, possible to better contextualize the principles with the following points to be pursued:

- *Traffic encryption*: The network, by its nature, must always be considered hostile regardless of whether it is internal or external to the organization or a cluster. Confidentiality, integrity and authentication should be accomplished for all communications permitting secure access to the resources and preventing data exfiltration from sniffing the network.
- *Traffic tracking and inspection*: All the traffic should be tracked and monitored. These actions permit to arise the awareness of the actual status of the network. Even if it may not be possible to perform application layer inspections, monitoring the traffic patterns and flow may help to detect anomalous behaviour. Regarding the inspection of the content of the traffic, it implicitly leads to a trade-off between encryption and monitoring. Techniques based on pattern matching in packet payloads and entropy analysis are an open field of research, based on machine learning, to provide traffic classification capabilities of encrypted traffic to Intrusion detection systems (IDS)[10]. Otherwise, the inspection is based on middleware capable of decrypting the traffic, analysing it and encrypting it again before sending it to its destination. Further considerations will be made taking into account the tools proposed for this thesis, but a first consideration to be made is that, in case it is necessary to have a complete traffic inspection, criticalities will necessarily be introduced in these inspection points.
- *Identity*: A strong identity for each connected resource is the first mandatory step to enable a Zero Trust model. Explicit authentication and authorization for each performed action are necessary, and it leads to the necessity of having a strong identity. First of all, identity is intended as the identity of the user, but a further step for a robust zero trust strategy includes also the devices: each device has to be uniquely identifiable denying access to anonymous devices and taking into account the devices' current security posture.
- *Control Plane and Data plane*: A clear distinction should be made between communication flows used for management and administration purposes and the ones used by tenants where most of the data are transferred to interact with, and between, applications and services. This separation may be logical, or even, better physical.
- *Granularity*: Assuming that a resource may always be compromised, the network segmentation and the micro-segmentation of security have to be applied to reduce as much as possible the attack surface area and impacts of lateral movements.
- *Scalability*: Points in which security controls are performed become key elements also in terms of performance and availability of the service. They have to be able to handle the incoming requests and scale rapidly accordingly to the traffic flow. Any delay or inability to reach a resource would determine the degradation of the service and must be avoided. This aspect also implies a trade-off, the security features must be implemented and will necessarily have an impact on the responsiveness of the service, which, however, must be acceptable and not impact the productivity.

These considerations are the basis on which we need to move towards zero trust. The following chapters will often refer to these concepts, showing tools that allow some of these measures to be implemented in practice in a cloud-native environment and specifically to mitigate the outlined open challenges of a Kubernetes cluster.

4.3 Zero Trust use cases

The zero-trust concept is a broad one, potentially applicable to a great variety of use cases and scenarios. The following are some examples that can give a glimpse of its wide applicability:

- *Enterprise with multiple offices and remote workers*: Geographically distributed organizations not directly connected with a physical network are one of the most common scenarios. Here, the principle of least privilege has to be strictly enforced. Due to the heterogeneity of connections used and devices, any credential provided to the employees may potentially be compromised and utilized for malicious activities. This concept is extended also to on-site visitors and contracted workers. Zero trust application can restrict their access only to the external network and, by default, deny any communication with enterprise resources.
- *Multi-cloud enterprises*: Data source hosted between different cloud providers, first of all for redundancy and resiliency reasons, can benefit from zero trust enforcing. For this use case, the NIST suggests moving the policy enforcement point close to each service and data source. It may become tedious given possible different security enforcements from the various cloud providers, but an overall compatible architecture has to be found to properly enforce zero trust principles.
- *IoT Devices*: Zero trust can even find applications concerning IoT devices. They introduce additional complexity due to heterogeneous hardware, software, design and deployment locations. At the same time, computational capabilities are also typically limited, not permitting the big overhead introduced by security features. Despite these limits, IoT devices are valuable targets to attackers and both Trend Micro[3] and Microsoft's blogs[4] provide insights into IoT zero trust application. This can be accomplished by segmenting the network in which the IoT devices are connected, generate data to assess the posture of the devices and, wherever possible, by providing secure connections between each other.

4.4 Zero trust in microservices infrastructure

Contextualizing further the zero trust principles presented in Section 4.2 into a microservices cloud infrastructure, leads to more detailed considerations. The nature of the microservices, or pods speaking about Kubernetes, is ephemeral. Both the microservices design and actual deployment is dynamic: it can span across multiple nodes and clusters. The latter can even be public infrastructure. Therefore, there is not a defined perimeter and security tools defined for static environments are no more effective. Different types of traffic also have to be considered and, according to the principles, all of them require a zero trust approach:

- *external traffic*: it is the most immediate one, just like traditional client-server architecture, it is the traffic coming from the client that has to be served. Just like any external traffic, it has to be considered hostile and strict security controls must take place.
- *internal traffic*: With microservices, in-cluster traffic is what permits all of them to communicate and actually generate the response. Following the zero trust approach, also this traffic must be carefully considered. Furthermore, assuming breaches, the microservices themselves must not be implicitly trusted.
- *egress traffic*: External traffic may also be generated from applications inside the cluster to access external resources and APIs.

All these considerations lead to new security approaches based on going towards zero trust. It does not exclude firewalls deployed on the edge of the environment, but it includes granular and flexible identities that permit authentication and authorization, traffic encrypted by default inside the whole cluster and observabilities tools to track this dynamic environment and be aware of its behaviour.

[3] https://www.trendmicro.com/zh_hk/research/21/i/iot-and-zero-trust-are-incompatible-just-the-opposite

[4] https://www.microsoft.com/security/blog/2021/05/05/how-to-apply-a-zero-trust-approach-to-your-iot-solutions/

Chapter 5

Service mesh and Istio analysis

This chapter introduces the theoretical concept of *service mesh*, an infrastructure level added on the top of Kubernetes that enables extra features capable of handle the open challenges presented in the previous chapter. Afterwards, it will dive into a specific service mesh named *Istio*, analysing the tool in its entirety. An important consideration is about the version used due to the fact that it is a fast-evolving technology and the project comes with a high frequency of releases. The entire thesis work is based on Istio `1.10.1`, any earlier or later version may have different functionalities and behaviours.

5.1 The Service mesh approach

Kubernetes main functionalities are about deployment. As previously described, several capabilities related to other aspects such as networking, logging and security may require the necessity of being implemented directly inside the application, embedding libraries and writing code. This is not ideal and deviates from the concept of microservice. One solution is about *API gateways*, such as *kong*[1]. The idea is about centralizing these functionalities and mediate all the interactions through the API gateway enforcing policies, traffic rules and so on. This still comes with its downsides in terms of bottleneck, overhead and single point of failure.
An alternative, that is maturing in recent years, is the *service mesh* approach. It takes the concept of decupling features from the application, but, based on proxies that work alongside the main container, they will be applied in a more distributed way: different controllers will be associated within each service and a global controller will just provide configurations updating and coordinating all the points of application. It leads to a separation of concerns between Kubernetes, which will keep its focus on deployment and management of container workloads at scale, and the service mesh, which cooperates to manage the network traffic in a secure, reliable and observable way.
Figure 5.1 illustrates this architecture, logically splitting it into a data plane and a control plane. The data plane is composed of the proxies, the elements close to the service in charge of handling the whole service traffic and control it. The control plane, on the other hand, manages and configures the proxies providing traffic rules, enforcing policies and collecting telemetry data. Traffic, therefore, does not pass through the control plane, but only between the proxies, appropriately configured. Another concept to note regarding the implementation of a service mesh is the transparency of this additional layer from the point of view of the applications. In most situations, they will not be aware of its presence and no changes need to be made in the implementation.

[1] https://konghq.com/blog/kubernetes-ingress-api-gateway/

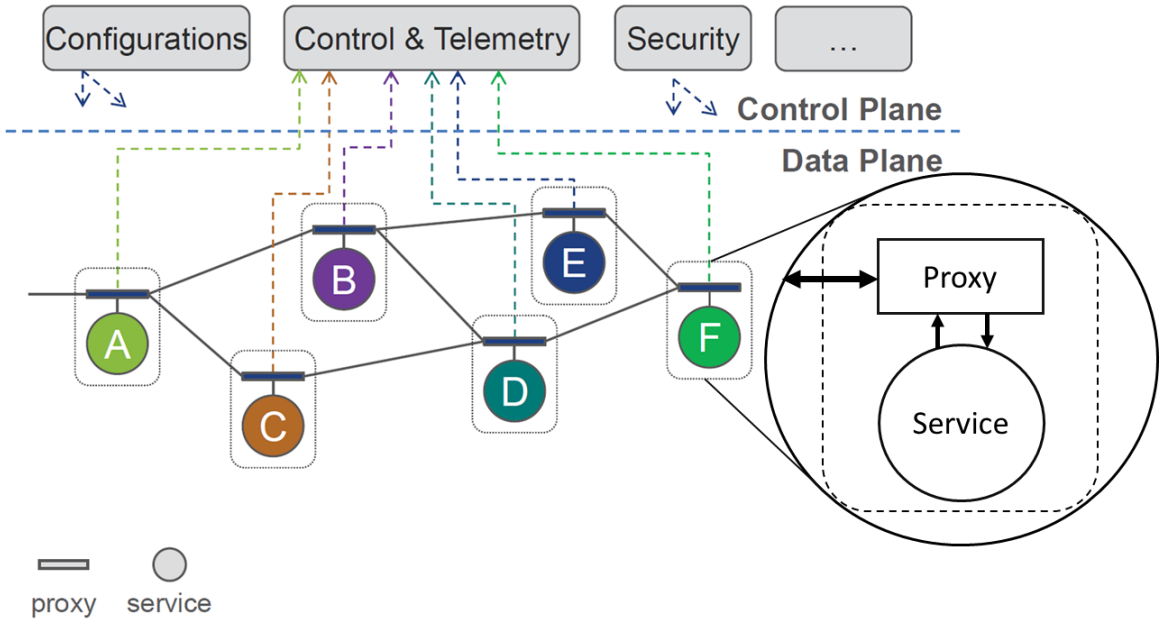

Figure 5.1: Architectural overview of the service mesh approach (source: [11])

5.2 Istio Overview

The open platform Istio matches precisely the service mesh model just described. Figure 5.2 shows a complete Istio architecture representation from the official documentation and the analogy with figure 5.1 is clear [12]. Each service interacts just with its own proxy. The latter, on the other hand, takes part of the Data plane and connects to the other proxies to actually send the traffic to any destination. As the figure shows, special proxies named Ingress and Egress are also part of the Data plane providing respectively connectivity from external networks and the possibility to establish connections to APIs located outside the cluster. The Istio core element is the control

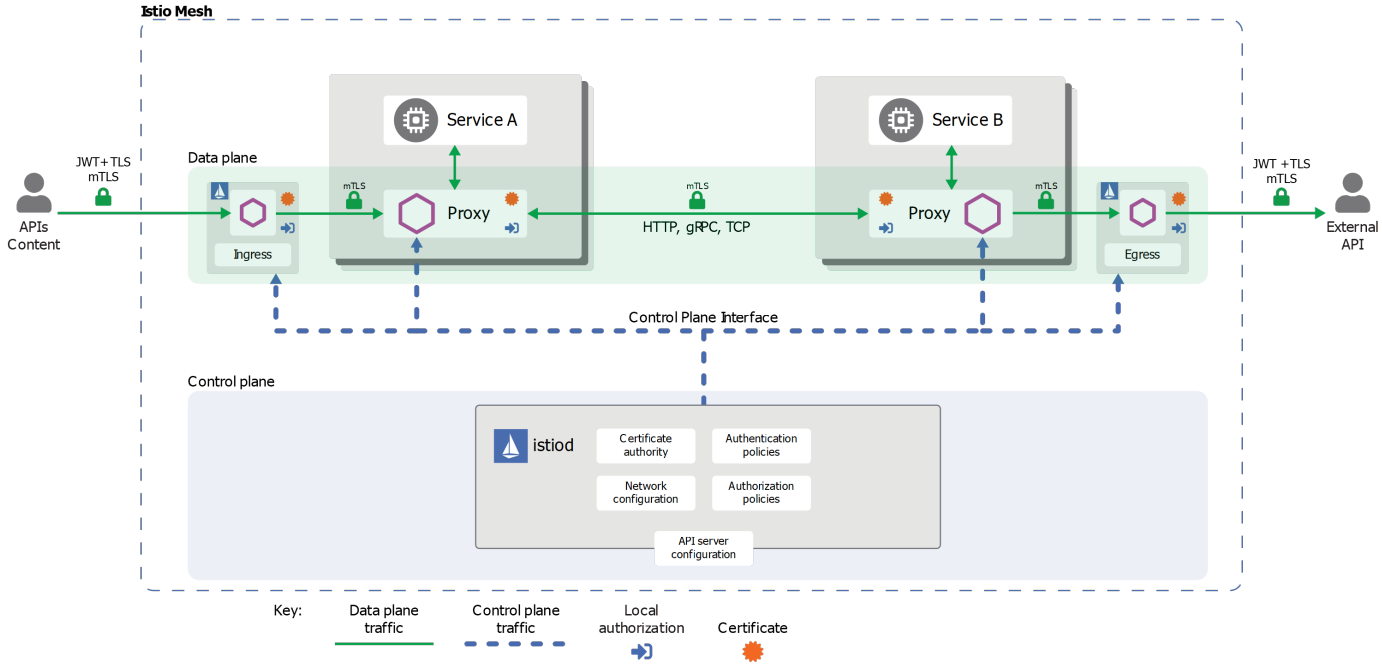

Figure 5.2: Istio Architecture overview (source: [12])

plane. Through it, all the proxies are injected, managed and updated, leading to a wide list of provided functionalities that include:

- Traffic management: The traffic routed between proxies can also be controlled with:
  - Traffic shaping with matching rules and weight-based traffic splits
  - Canary deployments
  - Circuit breakers
  - Request timeouts
  - Retries
  - Mirrored traffic
  - Fault Injection
- Observability: Traffic across the proxies is monitored, permitting to understand the behaviour and performance of the cluster. The customizable telemetry includes:
  - Detailed metrics
  - Distributed Traces
  - Access logs
- Security capabilities: Protection of microservices deployments are based on security-by-default functionalities and features that permit to go towards defense in depth approach:
  - Strong Identity for every workload
  - Peer authentication based on mutual TLS
  - End-user authentication JWT-based
  - Authentication and authorization policies
- Extensibility: The proxies can be extended with custom logic thanks to developed modules executed inside a sandbox. The goals of this extension technology are:
  - Efficiency: low latency, CPU, and memory overhead.
  - Isolation: each extension is executed isolated inside a sandbox.
  - Configuration: Istio control plane handle the dynamic configuration of the extensions.
  - Multiple languages available to write extensions

To provide these functionalities, Istio takes advantage of the modularity and extensibility provided by Kubernetes: via native resources such as pods and services, and custom ones (CRD, see section 3.3.7). Everything is integrated inside the cluster adding a new, fully compatible, layer in a transparent way. Istio configuration does also not require editing deployments and services, it is separated from the application configuration and based on YAML configuration files that will be handled by kubectl.

5.2.1 Data plane

The solution adopted by Istio is to implement the proxies as sidecar containers (see 3.3.1) by deploying *Envoy proxies* inside the pod of every application. Therefore, communication will never take place directly between the pods, but the traffic is routed to the sidecars and they will ensure networking and any other features. The set of these proxies forms the Istio data plane. Envoy is an open-source C++ proxy designed to be implemented in a distributed way at the edge, close to single services and applications [13]. By doing so, network functionalities are outside the logic of the application and independent of the application language used. Envoy provides a wide variety of functionalities extending the core feature of being an L3/L4 network proxy:

- A pluggable filter chain mechanism to write filters or apply some of the already implemented ones for common tasks and applications.
- HTTP L7 support both in terms of filter layer to perform tasks such as rate limiting and routing, and flexibility about protocols used at L7. It is indeed capable of bridge HTTP/1.1, HTTP/2 and HTTP/3 (currently in alpha stage) in any communication direction.
- gRPC support, widely used in distributed client-server architectures.
- A configuration API allowing different approaches for configuration management, from static ones to dynamic configurations.
- A health checking system to keep tracking the status of the pods. It can be active, sending requests and analyzing the response, or passive, based on outliers detection such as consecutive response errors or success rate.
- Advanced load balancing capabilities based on the availability of the hosts, specific application-level parameters, the possibility of choosing between different algorithms and via global decisions defined by a control plane.
- TLS termination and TLS origination to remove security duties from the main logic and, at the same time, being able to take advantage of unencrypted traffic.
- Observability via logging and statistics collection to monitor and debug network-level problems.
- Support for tracing via external tracers like Jaeger.

5.2.2 Control Plane

Envoy implements advanced features, but its scope is the single pod and, as described, pods are made to be ephemeral. The key characteristic of Istio consists in its ability to instantiate, manage, monitor and distribute configurations to all the dynamic Envoy proxies distributed across the cluster. This is accomplished by *istiod*, the core element of Istio's control plane. Istiod is composed of a set of pods and services inside a specific namespace called `istio-system`.
Istiod is capable of:

- Convert high-level written routing rules, typically given as YAML configuration files, into envoy configurations and then propagate them throughout the mesh to each Envoy sidecar.
- Manage authentication and authorization policies to enable service-to-service authentication, end-user authentication and access control for each workload.
- Act as Certificate Authority (CA): generates, rotates and revokes certificates to provide built-in identity and TLS encryption across the mesh.
- Collects telemetry.

Another element that is still part of the Control Plane is the *istio-agent* (also called pilot-agent). One istio-agent istance is executed inside each pod. It has the function of bootstrapping Envoy and permit it to connect to the service mesh providing configuration and secrets.

To recap, Istio, at a high level, can be seen as an Envoy proxies manager: it inherits their functionalities and centralises the management by adding configuration automation, flexibility and a set of services to take full advantage of Envoy.
First and foremost, to exploit the service mesh, the data plane must be in place: sidecars have to be deployed. Istio itself is capable of handling it providing a manual injection into the pod template via the Istio configuration command line (`istioctl`, but, above all, it permits to automate the injection inside each deployed pod. By adding a label (`istio-injection=enabled`) to a namespace, any pod deployment (in that namespace) will be effortlessly customized implementing

the sidecar proxy[2].
From the point of view of the functionalities provided by adding the Istio layer to the Kubernetes cluster, it is possible to divide them into three categories: traffic management, security and observability features. Reading an overview of these features, it is important to note the high dynamism introduced by Istio in performing these processes: everything can be easily adapted to the need of the moment just changing few configuration files and letting the control plane propagate the updated configurations to all the sidecars of the mesh.

5.3 Traffic management functionalities

As pointed out in section 3.5.1, Kubernetes lacks features for fine-tuning routing and granularly manage the traffic inside the cluster. Istio escalates the service discovery system of Kubernetes to populate a dedicated service registry learning all the endpoints of the cluster. At this point, via CRDs entities it is capable of route traffic between any service, load balancing it between multiple replicas or versions of the service, perform reliance and network tests. Furthermore, Istio traffic management permits to manage incoming and outcoming traffic via ingress and egress gateways providing all the traffic features also for external services, integrated them inside the cluster in a seamless way. Kubernetes itself provides a way to access external services defining services without selectors[3] but without this level of integration provided by the management that may be applied to this type of traffic. All these features are completely delegated from the developers to operations teams, that, via YAML configuration files, can independently manage the traffic of the whole mesh.

5.3.1 Traffic shaping

For traffic shaping Istio provides two routing primitives: *Virtual services* and *Destination rules*. Virtual services describe how the traffic has to be routed to a given destination via the creation of routing rules. By default, a round-robin algorithm is applied, but specific decisions can be taken based on matched rules. The latter can be expressed on traffic ports, header fields, URIs, prefixes, exact values or more complex regex matches with RE2 syntax. Figure 5.3 shows a virtual service example in which the content of the HTTP header field `end-user` is evaluated with a regex to make decisions about the route that the traffic will take. Specifically, if it ends with `@polito.it` the traffic will have as a destination the frontend service version 2. Otherwise, other match rules will be evaluated. In this case, no more rules are present, so, all the requests not matching the first one, will be forwarded to the default destination, written as the last route, without any prepended match pattern. In this case, the default destination is still the frontend service, but in its version 1.
After virtual service routing evaluation, destination rules are applied. Via destination rules is it possible to configure what happens to the traffic that is reaching a destination, define named service subsets and enable custom outlier detections to detect unhealthy pods. The example in figure 5.4 shows the specification of two subsets, `v1` and `v2`, and custom load balancing policies for the traffic reaching the two different version. For v1, the replica that will handle the request will be selected, between the health one, via the default round-robin. On the other hand, for v2, the decision will be taken randomly picking two hosts, and selecting the one with fewer active requests. Details about these, and other load balancing options, can be found inside the official documentation[4].

[2] https://istio.io/latest/blog/2019/data-plane-setup/

[3] https://kubernetes.io/docs/concepts/services-networking/service/#services-without-selectors

[4] https://istio.io/latest/docs/reference/config/networking/destination-rule/#LoadBalancerSettings-SimpleLB

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: frontend
  namespace: default
spec:
  hosts:
    - frontend
  http:
  - match:
    - headers:
        end-user:
          regex: (\W|^)[\w.\-]{1,25}@(polito)\.it(\W|$)
    route:
    - destination:
        host: frontend
        subset: v2
  #default route
  - route:
    - destination:
        host: frontend
        subset: v1
```

Figure 5.3: Virtual Service example

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: frontend
spec:
  host: frontend
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
    trafficPolicy:
      loadBalancer:
        simple: LEAST_CONN
```

Figure 5.4: Destination Rule example

5.3.2 Weighted Load balancing

Going further on load balancing, Istio provides a weight system for a manual specification of the traffic distribution rather than applying defined algorithms. This is a needed feature to perform *A/B testing* and *canary deployments*: tasks, that, with Kubernetes, could have been only performed working statically on the number of replicas deployed, implicitly determining the

```
- route:
  - destination:
      host: frontend
      subset: v1
    weight: 85
  - destination:
      host: frontend
      subset: v2
    weight: 15
```

Figure 5.5: Snippet of VirtualService for weighted distribution of traffic between two versions

ratio of each version of the service.

5.3.3 Network Testing

Features for networking testing, if used, previously were typically implemented at application level. Thanks to Istio now the following capabilities can be implemented at network level in a practical and effortless way:

- *Traffic mirroring*: Traffic can be duplicated mirroring it into another service. This is done to work on real incoming data with services still under development and testing, without impacting the main path that it should usually take.
- *Fault injection*: The reproduction of faults and latency problems permits to stress the system looking for unexpected and unhandled behaviours. It allows to anticipate and handle in time problems that most likely would have appeared later on during the production phase.
- *Circuit breaking*: Destination rules permit also to set circuit breakers: quotas about the traffic to individual hosts within a service that permits to anticipate severe performance problems and fail status. A fail-fast behaviour leads to reacting faster and redirect the traffic to healthy pods rather than keeping on contacting the problematic ones.
- *Timeout and retries*: By default, Istio does not enable timeouts and retries settings for HTTP requests, but rather, via virtual services, permits to dynamically set them on a per-services basis decoupling it, once again, from the application code.

5.3.4 Gateways

Traffic management means not only handling "east-west" traffic but also "north-south" made of inbound requests and external services that internal ones have to access. Istio provides *gateways*, implemented as envoy proxies, not running as sidecars, but standalone, at the edge of the mesh. Gateways can be:

- *Ingress Gateways*: To manage incoming traffic. An example can be seen in figure 9.1. Compared to Kubernetes Ingress resource, better customizations may be performed: the traffic will be managed like any other data plane traffic in an Istio mesh, so it is possible to fully apply all the Istio features on it such as routing and monitoring. For all intents and purposes, the Envoy proxy which forms the ingress gateway participates in the service mesh.
- *Egress Gateways*: To limit external traffic to go through it defining exit points from the mesh. Just like Ingress, it is possible to apply Istio features on them for a complete control of the traffic.

```
apiVersion:
    networking.istio.io/v1alpha3
kind: Gateway
metadata:
name: gateway-example
spec:
selector:
istio: ingressgateway
servers:
- port:
number: 80
name: http
protocol: HTTP
hosts:
- "example.k3s"
```

Figure 5.6: Ingress gateway example

```
apiVersion:
    networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: vs-gateway-example
spec:
hosts:
- "example.k3s"
gateways:
- gateway-example
http:
- match:
- uri:
exact: /main
route:
- destination:
host: frontend
port:
number: 9080
```

Figure 5.7: VirtualService associated to the ingress gateway

Figure 5.6 shows a simple config file that leads to deploy an ingress gateway. It defines the protocol used, the port on which the listener will wait for requests and the host name related. The traffic management is completed configuring also a VirtualService associated to the ingress gateway (see figure 5.7). Here, based on matches on the URI, the traffic is forwarded to the destionation workload that will compute the response. A more complete example of ingress gateway configuration is shown inside the appendix at section A.2.

5.4 Security functionalities

In terms of security, Istio has two objectives that perfectly fit a zero-trust approach:

- *Security by default*: Implementation of security features should be as much as possible enabled by default and transparent for the applications deployed, effectively by implementing it without the need of application code changes. Specifically speaking about microservice-based applications, it permits to decuple security implementation from application development reducing the effort for the developers and the risk of insecure implementations.
- *Defense in depth*: Security controls implemented at L3/L4 via Kubernetes capabilities (see Section 3.4) are totally compatible with the Istio additional security layer that can complement, or further enforce, already existing ones. Strong identity, policy enforcement up to L7, transparent mutual TLS encryption across the whole service mesh, authentication and authorization opportunely integrated with logs that permits to be aware of what is happening, are all capabilities that go towards a zero trust across distrusted networks. As the following sections deeper show, Istio permits all of this.

5.4.1 Strong Identity

The first feature presented is what leads to a strong identity definition for all the workloads. It is the basis for a security infrastructure in which actions have to be restricted to authenticated

and authorized entities. Istio, by default, provides for each workload an identity via an X.509 certificate. The provision is done via the following 5 steps that can also be seen in figure 5.8:

1. Via a gRPC service, Istiod can be called to request a certificate (certificate signing requests (CSRs)).
2. The istio-agent, inside each pod, at start-up, generates a private key and the CSR and sends the latter to istiod service.
3. The Istiod certificate authority (CA) validates the CSR credentials, signs and generates the certificate.
4. The certificate is received by the Istio-agent which will provide it to Envoy alongside the private key (via Envoy secret discovery service (SDS) API[5]).
5. Now an identity to the workload is provided. The Istio-agent monitors the expiration of the certificate and repeats the process periodically to update it and change the keys.

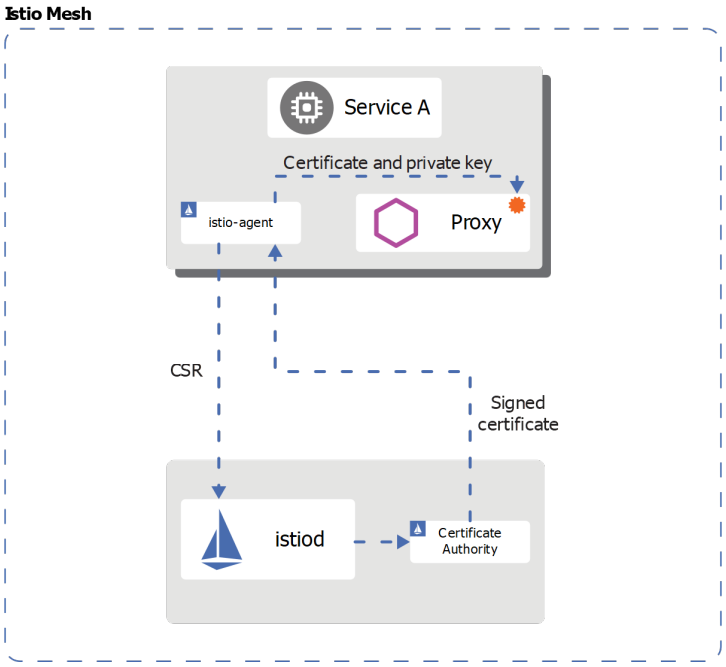

Figure 5.8: Istio identity certificates ditribution (source: [12])

It is a first needed step towards identifing and trusting the single entities and not the whole network. The interaction between each other will be based on a first exchange of credentials permitting a mutual authenticated connection.

5.4.2 Peer authentication: mTLS

Once an identity is provided for each entity, communication between each other can be secured preventing eavesdropping and man-in-the-middle attacks inside the service mesh. Mutal TLS (mTLS) permits to mutually authenticate the two parties verifying the other's certificate. There will still be a client, that starts the communication, and a server (another workload) that will handle the request. For the first, the identity received from the server is verified during the TLS handshake thanks to a secure naming lookup based on secure naming information provided by

[5] https://www.envoyproxy.io/docs/envoy/latest/configuration/security/secret

the control plane[6]: it means that the client ensures that the certificate just received is authorized to run the contacted service. From the server point of view, having a client verified identity it becomes possible to reliably provide audit logs and verify based on authorization policies if the client is allowed to perform that request. Istio, via the CRD *PeerAuthentication*, permits to set mTLS with three different levels:

1. *PERMISSIVE*: By default Istio configures sidecars to use mTLS, but still both plaintext and mTLS is allowed.
2. *STRICT*: Only mTLS is allowed.
3. *DISABLE*: mTLS is disabled.

These levels can be granularly configured both on namespace and workload level, but also apply more specific rules such as allowing plaintext from just one port of a workload. All these options come in handy when migration to mTLS of already existing workloads is performed. It may be needed a transient time in which both mTLS and third-party encryption are used, for instance, for workloads without a sidecar yet. Enabling a two-way authentication is a key element highlighted both inside the Kubernetes open challenges (see Section 3.5.2) and in zero-trust principles. Thanks to Istio, the establishment of a secure channel becomes transparent to the application and enforced, even if in permissive mode, by default. Section 9.2, inside the use cases chapter, provides a practical example of mTLS enabled between two workloads in its different levels. Concerning north-south traffic, it is expected that the traffic that comes from outside the cluster is secured. In this case, the ingress gateway will act as TLS terminator and, form that point, internally to the service mesh, mTLS will be performed up to the sidecar related to the requested microservice.

5.4.3 End-user authentication

Throught JWT tokens, Istio provides end-user authentication features, permitting to decouple also this element from the application code. It is possible to use a custom authentication provider or known OpenID Connect providers like Auth0, Keycloak or Google Auth. Via *RequestAuthentication* the end-user authentication can be required for namespaces or even specific workloads and specify the details about the token issuer and the public key that has to be used to perform the JWT validation. Figure 5.9 provides an example of an end-user authentication for the ingress gateway with JWT token issued by Auth0, while Section 9.4 provides further considerations appling the end-user authentication in a sample scenario.

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: "jwt-example"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  jwtRules:
  - issuer: "https://istio-auth0.auth0.com/"
    jwksUri: "https://istio-auth0.auth0.com/.well-known/jwks.json"
```

Figure 5.9: RequestAuthentication example applying end-user authentication to ingress gateway

[6] https://istio.io/latest/docs/concepts/security/#secure-naming

5.4.4 Authorization

Once each workload have an identity, authorization policies can be applied to specify what each identity can, or can not do. It is another pillar of zero trust: entities have not only to be identified, but each one has to interact just with the designed workloads (following the least privilege principle). These policies are configured via a custom resource named *AuthorizationPolicy*. The latter replaces the old CRDs named *ServiceRole* and *ServiceRoleBinding*. It can be applied at mesh, namespace or workload level. Four actions can be performed:

1. *ALLOW*: If an allow rule is matched, the request is allowed.
2. *DENY*: If a deny rule is matched, the request is denied.
3. *CUSTOM*: Custom rules are an experimental feature, it is used to delegate to a custom external authorization system (extension defined in the global options of the mesh) the decision of allowing or denying the request. Further details about it can be found in a recent Istio blog post[7].
4. *AUDIT*: Audit actions do not have an impact on the request, but rather define if the authorization request has to be logged. It only works if a supporting plugin for audit is enabled[8].

Excluding audit policies, the other actions are checked following these rules order:

1. Custom policies are evaluated, the request is denied if the external authorization system denies it.
2. Deny policies are evaluated, the request is denied if there is a match.
3. If there are not allow policies, the request is allowed.
4. If at least one allow policy is defined, evaluates them and the request is allowed if there is a match.
5. The request is denied.

It is therefore possible to note that custom policies does not allow directly the request. If used with other policies, they will also be evaluated before allowing the request. Authorization criteria can be defined based on a wide range of possibilities supporting TCP, gRPC, HTTP, HTTPS and HTTP/2 natively. For instance, it is possible to allow just specific HTTP methods for a specific workload, or deny any connections not coming from a defined namespace. These examples, with code snippets, are shown in Section 9.3.

5.5 Observability functionalities

Through the generation of telemetry regarding all the communications inside the mesh, a significant degree of observability of the service mesh is achieved. This provides a better understanding of how Istio, and all the microservices are behaving and interacting with each other. It permits monitoring, troubleshoot and proper management of the service mesh. Also in terms of security, zero trust principles highlight how important is to be aware of how the whole infrastructure is behaving to properly provide a suitable security posture and fast react if problems arise.

[7] https://istio.io/latest/blog/2021/better-external-authz/

[8] https://istio.io/latest/docs/reference/config/proxy_extensions/stackdriver/

5.5.1 Telemetry types

Three different types of telemetry are generated by Istio providing different levels of observability:

- *Metrics*: Divided into metrics at:
  - *Proxy-level*: Each Envoy proxy is capable of generating and sending metrics to Istiod about inbound and outbound traffic, pod health and the configuration status of the proxy itself. It is possible to configure the number of metrics generated to provide full visibility during development and testing, but then reduce the generation of metrics during the production phase balancing observability and performance.
  - *Service-level*: Based on latency, traffic, errors and saturation, Istio generates metrics at the service level to monitor how they are interacting with each other. By default, these metrics are recorded by Prometheus and can be queried or visualized with dashboards like Grafana. The following section further describes these tools. A default set of service level metrics is collected, but they are entirely optional. It is possible to customize the statistic configuration according to the needs of each environment[9].
  - *Control plane-level*: Made of self-monitoring metrics that track the behaviour and statistics of Istiod itself. For example, the total number of sidecar injections requested, skipped and successfully completed and the number of CSRs received, parsed, signed or ended with an error are collected. The full list of exported metrics can be found inside the Istio documentation[10].
- *Tracing*: Tracing telemetry permits monitoring from the point of view of a single request. Distributed traces are collected by all the proxies that the traffic flows through and can be analyzed via tracing backends like Jaeger. It is particularly useful in terms of performance analysis detecting bottlenecks.
- *Access Logs*: The point of view of a single workload is accomplished by access logs. Figure 5.10 shows a one-line access log example in which a `GET` request to the `productpage` workload is logged.

```
[2021-10-25T15:11:49.084Z] "GET /reviews/0 HTTP/1.1" 200 - via_upstream - "-"
    0 375 1332 1331 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36"
    "cf5b05b4-36af-96dd-ac22-e9aa64c60951" "reviews:9080" "10.42.1.18:9080"
    outbound|9080||reviews.istions.svc.cluster.local 10.42.1.20:50478
    10.43.182.73:9080 10.42.1.20:41348 - default
```

Figure 5.10: Access log example

5.5.2 Observability tools

Istio provides an effortless implementation of the following tools:

- *Kiali*: It is a service mesh management console, via a web-based GUI it is possible to graphically visualize the service mesh, details about the deployed resources, configurations, labels and security aspects like all the communication secured by mTLS. Some service metrics are also visualized providing the status of connections in a customizable time range. Figure 9.2 shows a view provided by Kiali, the graph of a small microservice application deployed, while figure 9.3 shows how a secure channel is displayed.

[9] https://istio.io/latest/docs/tasks/observability/metrics/customize-metrics/

[10] https://istio.io/latest/docs/reference/commands/pilot-discovery/#metrics

- *Grafana*: It is based on default and custom dashboards and permits monitoring the service mesh showing graphs and aggregated information typically provided by Prometheus. Several dashboards are already present to provide specific information about services, workloads, performance, control plane, wasm modules.
- *Prometheus*: Prometheus monitoring system permits to record metrics and queries them. Both Grafana and Kiali relies on Prometheus to retrieve the data that will be elaborated and visualized.
- *Jaeger*: Jaeger is one of the tracing backends supported by Istio. It permits to perform distributed tracing monitoring and troubleshooting from the point of view of a single request flow. Traces are shown as a set of spans. Each span is the time used by a specific workload to handle the request.

5.6 Istio Traffic capture limitations

Embracing Istio service mesh, expecially in terms of a security layer, does require to be aware of its current limitations and how to deal with them. The most significant is what type of traffic does the proxy actually intercepts: indeeed, it does not include all the traffic that the application container may generate. Istio supports all the protcols based on TCP: starting from the raw TCP, HTTP, HTTPS, gRPC up to experimental support of application protocol such as mongo, mysql, redis. The latter traffic, if not explicitly enabled, will be still proxies as raw TCP[11]. It means that any no-TCP traffic, such as UDP and ICMP, is not going to be intercepted by the proxy. Therefore, all the authentication and authorization rules, that are enforced at proxy-level, are not taken into account. It is shown in section : there, a deny-all policy is applied on the whole namespace. All the TCP traffic is correctly denied, but two pods, interacting with each other with UDP, had no restrictions. Furthermore, sidecars and application containers share the same network and process namespace. Potentially, from the container application, it may be possible to alter the sidecar behaviour. Here comes the defence in depth concept of zero trust: the Istio security layer should be further enforced by the underlying layer by properly setting Kubernetes' NetworkPolicies (see Section). The previously mentioned use case exploits them effecively blocking no-TCP traffic inside the namespace.

[11] https://istio.io/latest/docs/ops/configuration/traffic-management/protocol-selection/

Chapter 6

Diving deeper into Istio extensibility capabilities

Extensibility, in terms of providing customizable logic inside the sidecars, permits to meet a wide range of needs. It may come in terms of supporting new protocols, implementing custom logic or enhance already existing ones. The initial Istio native extension model proposed by the community (named Mixer) has been deprecated in favour of the WASM based extensions. Once introduced the WASM technology, this chapter aims to provide an overview of the viable extensibility strategies, better describe the WASM applied to Istio and dive deeper into how a request flows inside the proxies and where the custom extensions are applied.

6.1 WebAssembly

WASM, the abbreviation of WebAssembly, is presented as a "*portable bytecode format for executing code written in multiple languages at near-native speed*" [14]. It is a virtual instruction set architecture, a machine-agnostic bytecode designed to run on the Web high-performance applications and CPU-intensive tasks. Here resides the main difference compared to Javascript that is an interpreted language. It became a W3C Recommendation and the fourth standard language to run natively in common browsers[1]. The WebAssembly bytecode is just the outcome of a compiled program that can be written in most of the available languages including C, C++, Python, Go and Rust[2]. It permits high flexibility in terms of development. WebAssembly's use cases are mainly about, but not restricted to, execution inside the browser. Web platforms can take advantage of it for usages like encryption, peer-to-peer applications, computationally intensive clients, image and video editing and image recogition[3]. Despite the main design is about being executed on the Web, non-Web embeddings are also part of the high-level goals of this technology and are leading to a system interface standardization (WASI[4]). It can be applied for server-side applications, computation of untrusted code server-side and hybrid native apps on mobile devices. It is usually accomplished by executing it inside a JavaScript VM. One of them is the *V8 engine*[5], a high-performance JavaScript and WebAssembly engine realized by Google. The compiled wasm application is executed by a WebAssembly engine in a sandboxed execution environment that permits to accomplish the isolation goal.

[1] https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en

[2] https://github.com/appcypher/awesome-wasm-langs

[3] https://webassembly.org/docs/use-cases/

[4] https://wasi.dev/

[5] https://v8.dev/

6.2 Extensibility strategies

6.2.1 Extensibility via WebAssembly modules

Given the WASM capabilities not restricted to client-side solutions, Istio and Envoy communities worked on implementing WASM module inside the Envoy proxies. To accomplish this task, inside Envoy has been embedded the *V8 engine* previusly intoduced. Overall, the implementation of extensions via WASM permits:

- Take advantage of a growing ecosystem in terms of community and tools not limited to this specific usage.
- Program in multiple languages (currently C++, Rust, Go, Zig and AssemblyScript SDKs are available, but potentially WASM supports lots more languages).
- Execute the extension's code in a memory-safe sandbox inside the Envoy sidecar not affecting the Envoy process itself or other extensions execution.
- Deliver and load the extensions to the whole service mesh via Istio control plane at runtime without having to use modified Envoy binaries.
- Introduce minor latency, CPU, and memory overhead.
- Fully control the traffic enforcing policies and making modifications.

Figure 6.1 summarizes the high-level architecture of a WASM extension run by Envoy. The extension, executed inside a V8 sandbox, receives in chronological sequence, events from Envoy regarding the different phases of a connection. During each of them, for example at the beginning of the connection, or when the request headers are received, the extension may operate calling APIs that permits to read and modify the actual packet elements, perform external gRPC and HTTP connections, do log activities and rely on extra functions already available like CEL (Common Expression Language) and FlatBuffers libraries.

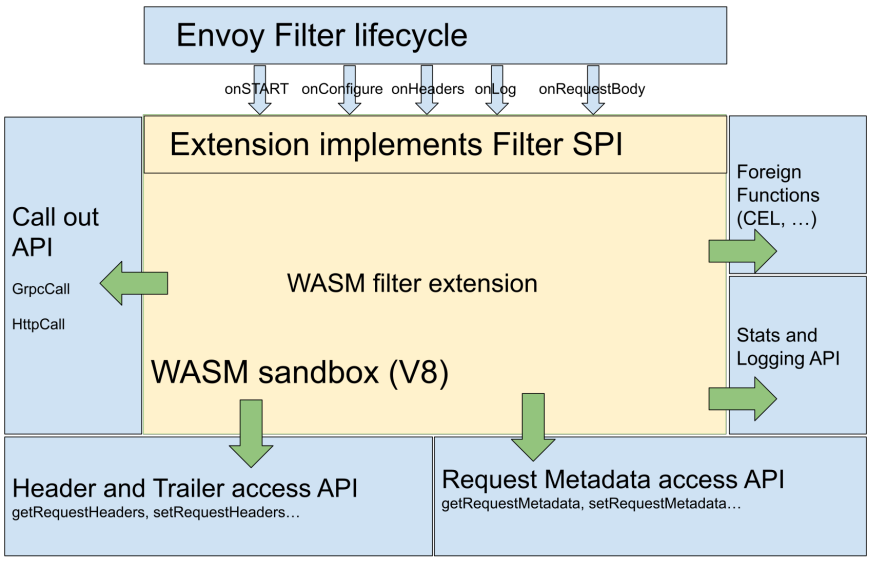

Figure 6.1: WASM extension High-level architecture (source: istio.io)

6.2.2 Extensibility via native Envoy

Alternatively to the extension made by WASM modules, has to be taken into account the Envoy extension model based on extending the monolithic proxy application. Comparing the two approaches, the latter:

- Requires less memory usage given the fact that VMs are not deployed.
- Overall better performances.
- Reduced binary size not implementing WASM runtime support.

On the contrary, directly extending Envoy:

- Requires to write the extension in C++ (Envoy's native language).
- Requires recompiling the whole Envoy and keep updating the customized version of it.

Given the described benefits and drawbacks, Istio is currently relying on WASM as its primary extension mechanism. Chapter 8 will introduce some already existing extensions provided as Envoy customization, but then the work will focus on exploring the current status of the WASM extensions and their capabilities in terms of deploying security controls in a flexible and granular way across the service mesh.

6.2.3 Extensibility via Lua scripts

A third approach, conceptually closer to the WASM approach, is based on implementing custom logic inline, directly inside configuration files using the interpreted scripting language Lua[6]. The execution is based on LuaJit, a Just-In-Time Compiler. It leads to better performances, but, being an independent implementation of Lua, it limits the support up to the features of Lua 5.1 (the last release was in 2012), with some exceptions from Lua 5.2, compared to the current version 5.4.3 released this year. Going beyond the smallest number of features compared to recent versions, also the features supported by Envoy comes with limitations in terms of the possibility of operating only on headers and trailers (the HTTP body is read-only). Given these considerations, Lua scripts inside Envoy are meant to be simple and safe to write, avoiding any complex or high-performance implementation. For the latter, it is recommended by the Envoy project to choose one of the other options and take advantage of Envoy's C++ API [13].

6.3 WASM modules distribution

Before running the extension within Envoy, the extension form must be provided to it. To accomplish this task, since Istio 1.9, it is possible to require the Istio-agent (see 5.2.2) to dynamically download the wasm file via two *EnvoyFilter* custom resources. The first one injects the filter into the proxy while the second one provides configuration, gives the reference from where to download the wasm file and links it to the just injected filter. The code snippets shown in Snippet 6.1 offer a more practical view of this configuration:

- Definition of the first EnvoyFilter and selection of the label of the specific pods that will receive the configuration (lines *1-8*).
- The WASM module is predisposed to work at the application layer (`HTTP_FILTER`) and is applied on incoming traffic (`SIDECAR_INBOUND`) (lines *10-13*).

[6] https://www.lua.org

- Definition of the second EnvoyFilter applied on the same workload (concordance with the one defined on the first EnvoyFilter is obviously needed) (lines *16-22*).
- This EnvoyFilter is set to provide configuration for the WASM extension (lines *24-27*).
- Declaration of the runtime engine (`runtime.v8`) and URI from which download the wasm binary (lines *29-34*).
- The sha256 of the binary is an optional element, but it is provided to allow the verification of the checksum from the Istio-agent. Otherwise, the module will be downloaded repeatedly skipping checking if it has been already downloaded and locally available (line *35*).

```
1  apiVersion: networking.istio.io/v1alpha3
2  kind: EnvoyFilter
3  metadata:
4    name: deploy-wasm-module
5  spec:
6    workloadSelector:
7      labels:
8        app: WorkloadA
9    [...]
10   configPatches:
11   - applyTo: HTTP_FILTER
12     match:
13       context: SIDECAR_INBOUND
14           [...]
15 apiVersion: networking.istio.io/v1alpha3
16 kind: EnvoyFilter
17 metadata:
18   name: deploy-wasm-module-config
19 spec:
20   workloadSelector:
21     labels:
22       app: WorkloadA
23       [...]
24   configPatches:
25   - applyTo: EXTENSION_CONFIG
26     match:
27       context: SIDECAR_INBOUND
28 [...]
29 vm_config:
30 runtime: envoy.wasm.runtime.v8
31 code:
32   remote:
33      http_uri:
34          uri: https://github.com/M4tteoP/wasm-repo/raw/main/basemodsec.wasm
35          sha256:
                05c2dc9e3fc836af8e9d5787ba84fb20ea8824dc2675a4c74fd76e5e7422a54f
```

Snippet 6.1: Snippets of the EnvoyFilters needed to run a WASM extension inside Envoy

A complete example configuration can be found here[7]. Figure 6.2 shows the flow starting from when the EnvoyFilter resources are added to the cluster. Istio-agent receives the configuration about the requested extension from Istiod, from there, based on the checksum, checks if the WASM binary is locally already downloaded, otherwise it proceeds with the download. Only if the WASM module is correctly retrieved, the configuration is distributed to Envoy as an Envoy

[7] https://istio.io/latest/docs/ops/configuration/extensibility/wasm-module-distribution/

Extension Configuration Discovery Service (ECDS) resource. Otherwise, the configuration will be rejected in order to avoid the loading of invalid configurations.

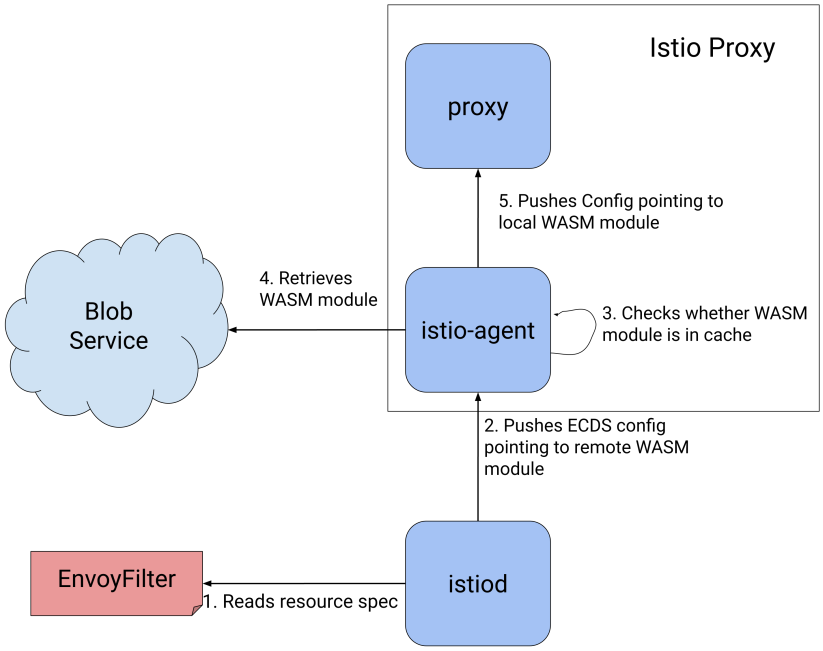

Figure 6.2: WASM extension High-level architecture (source: istio.io)

6.4 Tracing a request inside Envoy proxy

Exploring how Envoy internally behavies handling an incoming request permits to deepen understand how and where the custom logic behind a WASM extension is applied. Envoy thread is based on a single process made of the main thread, that handles configurations and the lifecycle of the proxy itself, and some worker threads that, via an event-based model, handle the requests in all their phases: listening, filtering and forwarding. From the point of view of a TCP request, the connection will be bound to a specific listener instance (inside a worker thread) avoiding any form of multi-thread complexity. The load balancing between multiple listening worker threads is performed at the kernel level. Once it is accepted by the listener, the connection goes through the first of three filter chains. These chains, graphically presented in Figure 6.3, are made by a set of modules (called *filters*) that, at different levels, accomplish some specific tasks on the connection. Most of the features enabled at a high level via YAML configuration files are translated in a combination of these filters to accomplish the required tasks. Each filter returns a filter status that describes how (and if) the chain will continue. One note to be done is about the temporal execution of the steps described below: as previously told, Envoy is event-based, so, a new set of chains is instantiated for each connection, and proceed independently depending on the order of the events, there are no blocking actions performed.
The first chain encountered is the *Listener filter chain* made of *listener filters*. They handle the raw data just received and can manipulate the connection metadata. Two examples are the *HTTP Inspector*, capable of detecting the version of the HTTP protocol used (and if actually is HTTP) and the *TLS Inspector* (shown in the figure). The latter discerns between plaintext or TLS and provides details about it. Then, if needed, the TLS transport socket performs the TLS handshake and provides the decrypted stream of data. The full list of listener filters already available inside Envoy can be found inside the project documentation[8].

[8] https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/listener_filters#config-listener-filters

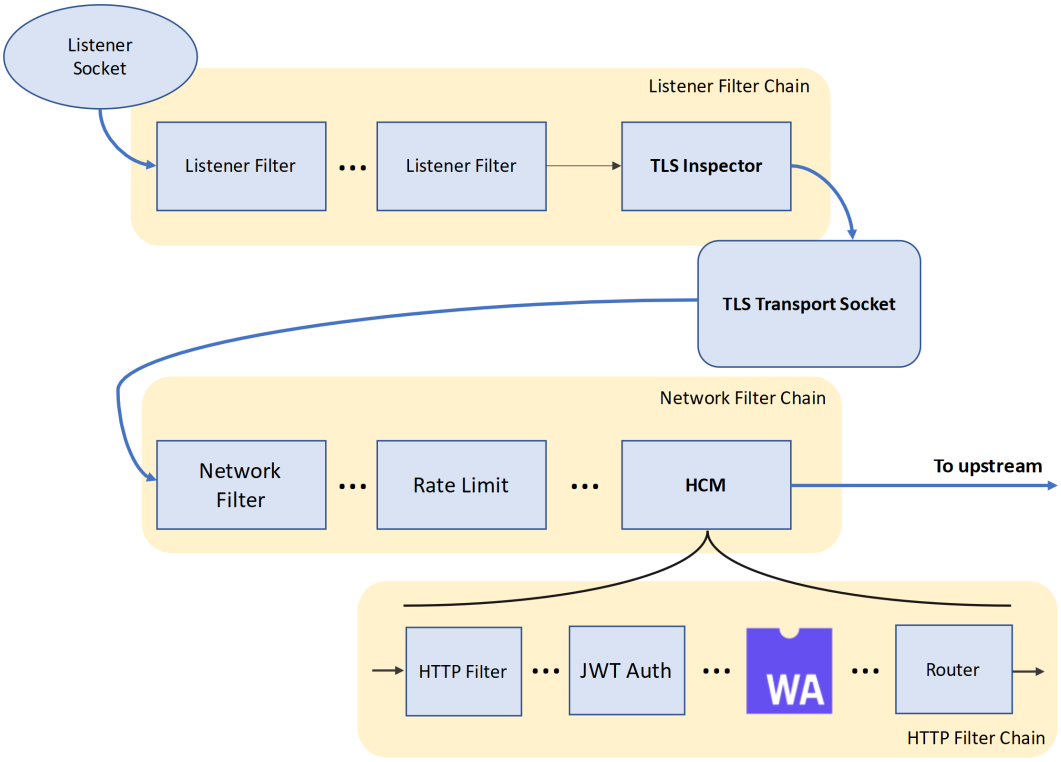

Figure 6.3: Filter chains inside Envoy proxy

Afterwards, the stream goes through the *network filter chain*. This chain is made of network-level (L3/L4) filters. Rate limit, Client TLS authentication and Role Based Access Control (RBAC) based on L4 properties such as IP and Ports are all examples of actions performed by these filters. The full list is available at the footnote url[9]. Here resides also the *HTTP connection manager* (HCM), a key filter to perform actions at HTTP application level. If necessary, HCM performs decoding and demultiplexing of HTTP/2 connections. The result is the creation of a set of HTTP streams. For each of them, HCM creates and manages the third and last chain called *HTTP filter chain*.
Via HTTP filters it is possible to read and manipulate HTTP requests and responses. Each HTTP stream will be populated with HTTP elements as soon as they are available. As a matter of example, when HTTP requests headers are ready, they will go through all the filters. Each of them will be capable of reading and manipulating them. Afterwards, if permitted by the last filter, the headers reach the next filter. A wide number of filters may be applied to enable features like fault Injection, end-user authentication (JWT) and RBAC based on L7 information, but, above all, here the WASM extensions described in Section 6 are applied. Just like any other already built-in filter, WASM modules are added to the chain and participate in performing actions on the HTTP streams. For the full list of available HTTP filters refer to the documentation[10] From the above example, it can be seen that the order of the filters is not irrelevant. For this reason, configuring the EnvoyFilter resource to deploy the WASM extension, is also needed to specify the insertion position relative to another filter. The snippet in Figure 6.4 shows the `operation` value equal to `INSERT_BEFORE` in respect to the `subFilter` defined above, the HTTP filter named `router`. The `operation` field can assume other, self explicative, values such as: `INSERT_AFTER`,

[9] https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/network_filters/network_filters#config-network-filters

[10] https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters#config-http-filters

```
[...]
 - applyTo: HTTP_FILTER
  match:
    context: SIDECAR_INBOUND
    listener:
      filterChain:
        filter:
          name: "envoy.filters.network.http_connection_manager"
          subFilter:
            name: "envoy.filters.http.router"
  patch:
    operation: INSERT_BEFORE
    [...]
```

Figure 6.4: Snippet of EnvoyFilter specifying the insertion position

`INSERT_FIRST` and `REPLACE`. A full list can be found inside Istio official documentation[11]. The HTTP filter router, shown in the example, is the last filter of the HTTP filter chain. It is the one in charge of selecting the route to reach the upstream endpoint and, if enabled, handling timeouts and retries features.

[11] https://istio.io/latest/docs/reference/config/networking/envoy-filter/#EnvoyFilter-Patch-Operation

Chapter 7

Web Application Firewall and ModSecurity project

This chapter describes the Web Application Firewall (WAF) introducing the purpose and detailing ModSecurity, the most widely known and used open source WAF. Later on, this knowledge will be applied in Chapter 8 in which a security control based on ModSecurity is proposed and developed escalating the Istio extensibility capabilities.

7.1 Introduction to Web Application Firewall

A WAF, as the name states, is a firewall that operates at layer 7 of OSI model providing security features via monitoring, filtering, manipulating and blocking packets directed to, and originated by, a web application. The application of a WAF permits to prevent the exploit of certain vulnerabilities present in the application itself by applying a set of rules. The rules may deny/allow packets:

- Based on information like the IP address, the HTTP methods used to perform the request and other specific elements inside the HTTP header. It permits to reduce the attack surface accepting only traffic potentially legit and reducing the risk of unexpected behaviours.
- Fully inspecting the traffic looking for patterns that may be related to common attack types. It can be typically performed both on headers and bodies (payload inspection) analyzing the potentially malicious user inputs and response pages generated by the application that may show typical patterns of information leakage. The main focus is to mitigate the most critical security risks of web applications.

A comprehensive list of web applications risks is provided by the OWASP (Open Web Application Security Project) foundation[1], which is also in charge of periodically update the so-called *OWASP Top 10*[2]. The latter document describes the most critical web application's security concerns of the moment, with a detailed description and remediation guidance for each one. The last OWASP Top 10, based on collected data dating from 2017 up to November 2020, has been published in 2021 and ranks at the first places broken access control, cryptographic failures and injection such as SQL Injection (SQLi) and cross-site scripting (XSS).
The importance of deploying WAF solutions is determined by the nowadays large diffusion of web applications and relative threats. The deployment fits a proxy-based solution in which the traffic flows before and after reaching the web application.

[1] https://owasp.org/www-community/vulnerabilities/

[2] https://owasp.org/Top10/

7.2 ModSecurity

ModSecurity project [15] has been initially released in 2002 to support Apache HTTP Server. It is a rule-based WAF with the ability to work both with a negative security model, explicitly denying traffic via a black list of rules, and in a positive security model (whitelist) defining elements (such as IPs, methods etc.) that the traffic must fit [16]. From the first release, several reworks have been done in order to permit an easier integration with other web services. Currently, two versions of ModSecurity are under maintenance: version 2.x, developed for Apache 2.x, and version 3.x that, via already available connectors, permits to integrate the WAF also with Microsoft IIS and Nginx. The latter version key element is not much about the extended support for two more common web servers, but rather a complete rework of the project releasing it with a new modular, and platform-independent, architecture. The result of this work is a C++ library (*Libmodsecurity*) that can be used not only with few available middlewares (called *connectors*), but also potentially integrated with any other web service. This section provides common elements of both versions, further specific considerations related to Libmodsecurity will be presented in chapter 8.

7.2.1 Functionalities

ModSecurity functionalities can be divided into four different areas:

1. *Parsing*: ModSecurity tries to interpret as much data format as possible in order to leverage all the possible information to make security decisions.
2. *Buffering*: Ideally, ModSecurity is suitable for buffering the whole request, analyse it in its entirety, and later on send it to the web application. The same approach will be taken for the response before passing it to the client. Given the necessity of storing data, it is a resource (RAM) demanding feature, but also an important one to provide reliable analysis of the traffic.
3. *Logging*: Modsecurity provides detailed logging features both in terms of debug logs to monitor the overall behaviour of the WAF and audit logs to collect information about all the traffic blocked. It is useful to raise the consciousness of attack patterns that the web application has to deal with, but it also introduces drawbacks in terms of performance and disk space usage due to the relevant sizes that log files may reach. Both logs can be customized enabling/disabling them and setting the verbosity or relevance thresholds.
4. *Rule Engine*: The engine is the component in charge of inspecting the collected data looking for matches with the loaded set of rules and, if necessary, take action.

7.2.2 Processing phases

As shown in the diagram in figure 7.1, the whole lifecycle of managing a request and the relative response goes through five phases. Each phase is in charge of taking into consideration a different part applying phase-specific rules.

1. *Request Headers*: The first phase is about initially assess a request based on initial information that can be retrieved by the HTTP Header looking for malicious patterns. It includes client IP address, URI path and translation of possible arguments in it and so on. Given the fact that after this phase it may be aspected to receive the body request, any preliminary actions about configuring how to (and if) process it must also be performed here.
2. *Request Body*: At the second phase, the body of the request is received. Now the whole request is ready to be analyzed in its entirety. It is the main request analysis step and it is a costly phase.
3. *Response Headers*: Once the request has been analyzed and, if allowed, sent to the server, the next elements that become available are the response headers sent back by the server. This phase is mainly used to make decisions about analyzing (and buffer) or not the response body that will arrive.

4. *Response Body*: Here the whole response can be analyzed looking for any form of information leakage that may be in form of default error messages, system versions and other common output patterns. Given the typically considerable size of response bodies, this phase must be carefully configured to avoid severe performance impacts.

5. *Logging*: The last phase does not permit any more to block the traffic, it is performed after having fully managed the request and relative response. The rules applied to this phase determine how the logging of the transaction just handled has to be done.

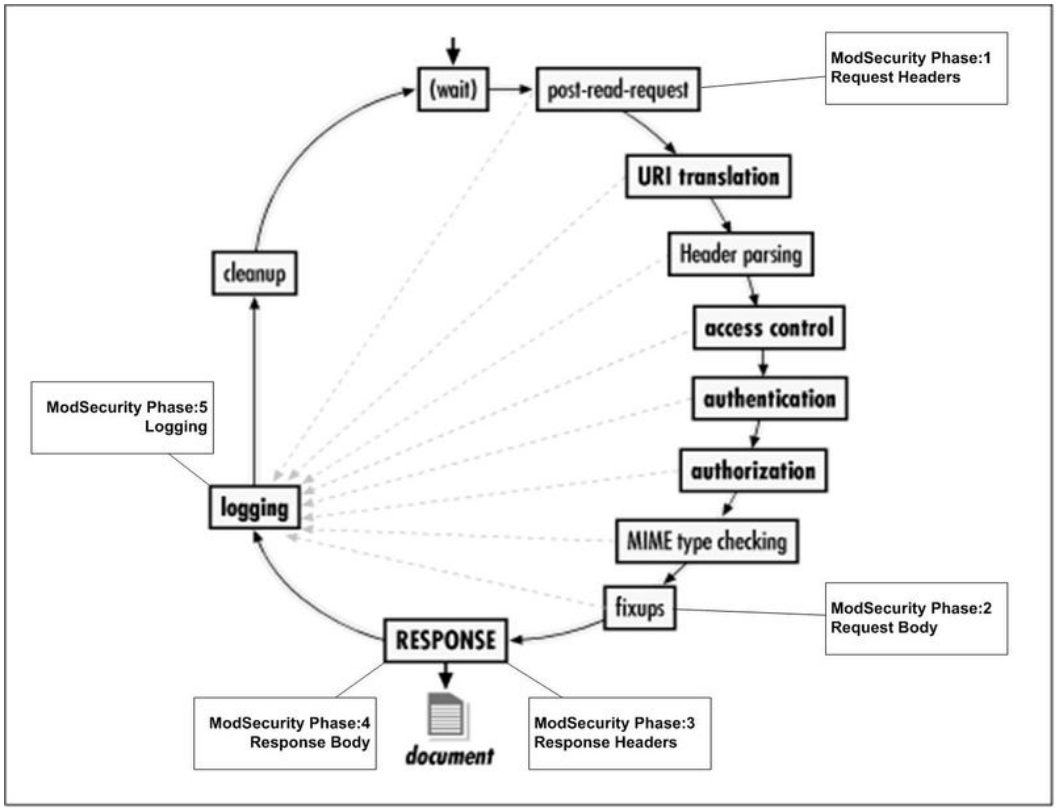

Figure 7.1: Five processing phases of ModSecurity (source: [16])

7.2.3 Configuration

An important guiding principle behind ModSecurity is flexibility: the WAF is virtually totally configurable, minimising anything performed implicitly and providing a powerful rule language capable of describing complex actions and performing them where it is needed. Behind this principle resides the willingness of providing a tool capable of adapting and working in most circumstances with the right setup. Above all, configuring and explicitly telling the tool how it has to behave leads to a better consciousness of the actions that you may aspect, of the responsibility of properly setting them and overall better security. Another consideration is about the impact on applying configurations and rules: security comes with a cost in terms of computational resources and availability of the service. Wrong configurations may lead to overhead that can not be handled or blocking traffic originated by legitimate requests.
Configurations are typically collected inside a file called *modsecurity.conf*. The main configuration file itself can still include several other files to create a more structured splitting of configurations. The list of all the configuration directives is extensive and can be found inside the documentation[3], the following list aims to give a glimpse of the main ones to familiarize with the basic configuration used in this work and with its syntax.

[3] https://github.com/SpiderLabs/ModSecurity/wiki/

- *SecRuleEngine*: It is the main directive, setting SecRuleEngine to `On` ModSecurity is enabled to process rules. Via `DetectionOnly` value it is also possible to enable the WAF without permitting it to intercept transactions. This deployment mode is suggested for initial tuning permitting to analyze the detections without affecting the traffic.
- *SecRequestBodyAccess* and *SecResponseBodyAccess*: As previously stated, it is possible to take decisions about analysing or not the request and response bodies. It has implications both in terms of security and performance. These directives enable or not this possibility. If they are disabled, any rule that should analyze the bodies at phases two and four will not work.
- *SecRequestBodyLimit* and *SecResponseBodyLimit*: Further configurations can be done about bodies in terms of limiting the size of accepted buffering.
- *SecRequestBodyLimitAction* and *SecResponseBodyLimitAction*: Reated to the previous directives, it is possible to decide what to do it the limit is reached: `Reject` value will drop the request, `ProcessPartial` will take into consideration only the information memorized up to the limit.
- *SecDataDir* and *SecTmpDir*: Directives respectively to tell where to store persistent and temporary data inside the file system.
- *SecDebugLogLevel*: Accepting values from `0` (no logging) up to `9` (log everything in a detailed way), it is possible to decide the verbosity of debug logs.
- *SecAuditEngine*: Directive to manage the audit logs enabling them for all transactions (`On`), just for the ones that triggered some rules (`RelevantOnly`, or disabling at all (`Off`).
- *SecDefaultAction*: Each rule may explicitly provide actions that must be done in case of a match, but this directive permits defining the default one.

7.2.4 Rules syntax

Configuration directives explicit how the traffic must be processed, while the rules decide the patterns that have to be detected inside it and how to react. `SecRule` directive defines a rule with the following format:

```
SecRule VARIABLES OPERATOR [ACTIONS]
```

Where:

- VARIABLES: They are the HTTP elements that the rule will inspect. Some examples are the arguments (`ARGS`), the request URI (`REQUEST_URI`) or the whole response body (`RESPONSE_BODY`). More of them can be concateneted via the pipe character "|" to perform the same inspection on more than one element.
- OPERATOR: It specifies the pattern that will be checked and the methodology. It starts with "`@`" and may be a complex regex, comparisons a fixed string or many other options. Refer to the documentation for the exaustive list of operators[4].
- ACTIONS: Define what to trigger in case of the rule matches. The categories of actions contain, but are not restricted to, disruptive, non-disruptive and logging The latter permit respectively to trigger an action that usually leads to a blocked transaction, execute something that does not alter the processing flow such as changing a variable and alter how the logging of that rule is performed. If no actions are provided the default one will be applied (see `SecDefaultAction` at Section 7.2.3).

[4] https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual-(v2.x)#operators

One elementary rule example is the following:

```
SecRule ARGS "@rx admin" "id:100,phase:2,deny"
```

Here a regular expression operator (`@rx`) is executed looking for the string `admin` inside the arguments. The action tells that this check will be performed at phase 2, where the whole request is managed by ModSecurity. This is done in order to do not consider only arguments present in the header, but also the ones coming from the POST method that resides inside the body. If the regular expression matches, a disruptive action (`deny`) is performed terminating the transaction. The numerical id of the rule is also present inside the action, it became mandatory from ModSecurity v2.7.0 and it is used to univocally refer to the specific rule. `SecRuleRemoveById` is a directive that may be used in conjunction with it. Providing a numeric identifier to this directive, the rule with that id will be excluded from the chain of rules. It becomes useful when big sets of rules are loaded, but specific ones must be disabled.

7.3 OWASP Core Rule Set

ModSecurity by itself is a WAF engine, capable of inspecting the traffic and perform actions. Actions that, as shown in the previous section, must be explicitly declared via rules. It is, therefore, necessary to provide a collection of them. Writing exhaustively would result in a prohibitive task, most likely ending it by do not take into account several attack patterns. For this reason, various collections of rules have been developed, some commercial, sold by security solution companies such as Trustwave[5] (The company itself supports the ModSecurity project until July 2024, and then will give the maintenance of ModSecurity code back to the open-source community), others publicly available. The most famous of the latter category is the open-source *OWASP ModSecurity Core Rule Set* [17], usually abbreviated as *CRS*. ModSecurity has become so prominent in the protection of web applications that the OWASP foundation has developed and periodically updated this rule set in order to define the first line of defence based on a set of generic attack detection rules. The stable version used in this thesis is the release version 3.3.2, but also a development branch (v3.4) is present and under active development to continuously address new threats and security concerns that may arise from the ModSecurity development.

Attack types detected

Implementing the CRS, ModSecurity becomes capable of handling a wide range of attacks and, at the same time, limiting as much as possible false positives. Currently, rules are divided into 13 different groups and include, but are not restricted to, OWASP Top Ten providing protection against:

- Crawlers, port and environment scanners
- Common web server application misconfigurations
- Protocol anomalies, violations and encoding issues
- Header injection, request smuggling, and response splitting
- File and path attacks
- Remote file inclusion (RFI) attacks
- Remote code execution attacks
- Injection attacks (PHP-injections, XSS, SQLi)

[5] https://www.trustwave.com/

- Session-fixation attacks
- JAVA attacks

The benefits of applying the CRS resides in the detection of these wide attack patterns that, furthermore, is performed directly in (or close to) the server, without any intervention of third party WAF providers. The patterns may also be properly customized adding custom rules for the specific usage and setting the *paranoia level* accordingly to the needs. It has four different values that can be summarized in the following way:

- PL1: The default value, suitable for generic sites and applications. It provides baseline protection.
- PL2: Extra rules are added mainly against obfuscated attacks, suitable for high security requirements.
- PL3: Includes rules for less common attacks and limits special characters to handle unknown attack types and further obfuscation techniques.
- PL4: Restricts the permitted usage of special characters even more

The drawback of elevating the paranoia level, and more generally of using the CRS, is about false positives. Even if minimizing them is a priority for OWASP, pattern detection will necessarily lead to some legit requests being blocked. Regarding this, it is suggested to properly monitor the traffic and handle the specific rules that may create problems excluding them. Given the number of rules applied, another drawback that has to be pointed out is a minor overhead of the processing.

Detection modes

The CRS can work with two different modalities:

1. *Traditional Detection Mode*: It is the expected mode in which a set of rules can work. Each rule is independent and as soon as it is triggered, the relative action is performed. There is not a shared knowledge based on previous partial results made by the WAF. It is the easiest way to run rules and, due to the fact that as soon as a disruptive match takes place the action is performed, there is an overall smaller introduction of latency. It also comes with downsides in terms of logging, only the first matched rules that enabled the action is logged, and in terms of cumulative rules severity. Low severity rules may not cause any disruptive action, but several of them triggered in the same transaction should have been taken more seriously: this can not be done in traditional detection mode.
2. *Anomaly Scoring Mode*: It is the default mode from CRS v3.x and permits to accomplish more complex considerations based on rules in their entirety, just like the example just made. This takes the name of *collaborative detection*, each rule match will not directly trigger the related action, but rather will participate in incrementing *transactional anomaly scores*. Reached certain, customizable, thresholds, actions will be executed.

```
severity:'CRITICAL',\
setvar:'tx.xss_score=+%{tx.critical_anomaly_score}',\
setvar:'tx.anomaly_score_pl1=+%{tx.critical_anomaly_score}'
```

Figure 7.2: Snippet of a SecRule action for anomaly scoring mode

Figure 7.2 provides a snippet of the actions related to a CRS rule that looks for an XSS pattern. It requires to increment two transaction variables, respectively named `anomaly_score_pl1`

and `xss_score`, by a `critical_anomaly_score`: a configurable value that marks the triggered rule as a critical one. This modality permits to have higher flexibility in terms of decisions taken. They may be the result of a multiplicity of small anomalies not considered individually particularly relevant or of even just one severe match (falling into the case of traditional detection mode).

Chapter 8

Design and Implementation of a security control using WASM Extensibility

Initial chapters introduced zero-trust principles, the importance of being applied in a distributed environment like a Kubernetes cluster, and how it is possible to satisfy some of them via available features provided by the Istio service mesh. Still, zero-trust is a concept to move towards, it is a process, not a final destination that may be actually reached. This is the reason behind the presentation of the extension's capabilities provided by Istio in synergy with the data plane made of Envoy's proxies done in Chapter 6: extending these elements can provide highly integrated security controls capable of further enforcing a zero-trust model and being flexible enough to the needs of various environments. Chapter 7 introducing the WAFs, and, specifically, what ModSecurity is capable of, was anticipating this chapter, in which is proposed the design and implementation of a WASM extension security control capable of perming WAF operations accomplished via the ModSecurity library.

8.1 Related works

Looking for solutions that implement security controls inside the service mesh, three categories, divided by the implementation approach of the solution, are found across the Web.

8.1.1 Envoy custom builds

These solutions follow the extensibility approach described in Section 6.2.2 based on rebuilding a customized version of the proxy enriched with extra capabilities and then providing it to be used instead of the native one. Projects taking this path are:

- *Glo Edge*: Presented as an API gateway, Glo Edge sits in the control plane and provides configurations to the data plane made of Envoy with an enriched set of built-in filters. In its commercial version, called *Gloo Edge Enterprise*, comes with some security controls that can be enabled in form of WAF, data loss prevention (DLP) and vulnerability scanner[1]. The operation mechanism is quite straightforward: the customized data plane is capable of interpreting Gloo objects (CRD in Kubernetes) that provide configurations to the custom Envoy. Requiring specific features, the relative filters are added to the filter chain.

[1] https://www.solo.io/products/gloo-edge/

- *ModSecurity-envoy*: The project found on GitHub, has been developed by Octarine startup, but, since the acquisition by VMware, it has not been maintained[2]. The custom version of Envoy includes an optionally available WAF based on ModSecurity.

8.1.2 WASM modules

The availability of modules built as WASM filters reflects the recent nature of the technology. The following are the projects found on the Web:

- *Istio-ecosystem*: This repository provides a list of already available WASM extensions projects[3]. They are didactical extensions made to provide development patterns and starting point projects. In terms of security control there are the following basic implementations:
  - *Basic Auth Filter*: The filter is capable of intercepting the request headers and checking if the request is authenticated verifying the host field and a custom header made of base64 encoded credentials or `username:password` pattern.
  - *Local rate limit filter*: Made of a token bucket with a ticker that refill it and request headers (the beginning of a new request) that consume the tokens. If no tokens are available che filter chain is interrupted sending back a HTTP 429 Too Many Requests error code.
  - *Open Policy Agent client*: The Filter takes decisions about allowing or not each request based on a HTTP call to a Open Policy Agent (OPA) server internal to the cluster.
- *WAF WASM filter*: Project on GitHub, shows a more complete, yet basic, extension[4]. The filter analyzes incoming requests and performs SQLi detections based on *libinjection*, a common SQL / SQLI tokenizer parser analyzer also embedded inside ModSecurity.
- *Webassembly Hub*: It is a service for sharing already compiled Envoy-based WASM Extensions, potentially ready to be directly added via EnvoyFilter[5]. Modules labelled with the security tag are few and about custom authentication filters based on header parameters or external calls.

8.2 Technological stack choices

Related works search highlights two aspects: first of all, there is an interest in solutions capable of enforcing security controls inside the service mesh and, more generally, an interest in enriching with security capabilities the Envoy proxy, for its many possible use cases. In addition, the solutions provided by developing a modified version of Envoy are way more complete compared to WASM alternatives, even resulting in commercial products. It is predicable, Envoy is an open-source software since 2016, developed with an established programming language like C++ is. Support for WASM extensibility has been introduced just since 2020, with WASM itself still young and with its main development focused on web solutions. These considerations, together with the comparison made in Section 6.2, show a yet to explore technological approach, but with potential and growing interest. It was therefore decided to develop the security control as a WASM module. Speaking about the security control to be implemented, Istio itself, as shown in Section 5.4, provides features like peer authentication and JWT authorization. Going beyond simple examples, it is necessary not to reimplement solutions already available. The security control picked has been a WAF based on Modsecurity: it is a well-established solution, with some already made implementations with Envoy that testify its value and benefits on adding it inside

[2] https://github.com/octarinesec/ModSecurity-envoy

[3] https://github.com/istio-ecosystem/wasm-extensions

[4] https://github.com/dengyijia/WAF-wasm

[5] https://webassemblyhub.io/

a mesh of highly interlaced web applications. Specifically, also CRS will be integrated to provide ModSecurity with its characteristic set of rules. In terms of programming language, despite WASM architecture potentially permitting various alternatives, the decision of a WAF based on Libmodsecurity library restricts the choice to the library programming language: C++. A final choice and design goal is about not altering the high configurability possibilities and dynamicity provided by all the above-mentioned elements. A WASM module is potentially integrable with any Envoy-based proxy at runtime just by providing the proper configuration with a reference to the module binaries to the control plane. Leveraging the default resources provided by the underneath Kubernetes and Istio, it permits to deploy the WAF granularly and dynamically wherever an Envoy proxy is used, be it a sidecar inside a pod, or deployed as a gateway. Finally, by keeping also Modsecurity's concepts about flexibility in its configuration, the design has to allow it, potentially providing a custom configuration for each deployed WAF instance.

8.3 Design of a WASM security control

Extensibility based on WASM modules potentially permits lots of possibilities in terms of security controls. The modules would be applied at runtime, even per-workload basis, directly inside the proxies. It is the same enforcement point of built-in authentication and authorization policies. Furthermore, these modules could easily rely on external hubs, just like the presented *Webassembly Hub*, that can permit to share and use modules built by the community directly via configuration files. It can potentially lead to a great variety of solutions: starting from simple rate limiters and custom authentication mechanisms up to intrusion detection systems and firewalls. They, consequently, would permit an effortless, flexible and high customizable enforcement of security controls across the service mesh. Based on the technological stack choices made inside the previous section, possible limitations that could have technically limited the design of a WASM security control have been searched. The following ones were raised:

- Being WASM a specific binary instruction format, everything has to be built for this architecture: the application itself that provides the logic of the module, but also all its dependencies. C++ building is based on the *emscripten* compiler toolchain[6]; it provides built-in support to just some standard libraries (libc, libc++ and SDL)[7]. Any other library has to be ported to WASM architecture. It is therefore needed to manually build and link it as a static library to the final WASM object.
- The execution inside the sandbox comes with restrictions in terms of access to the operating system including all file systems operations, performing network operations, executing the `sleep` function. Basically, it is mandatory to rely exclusively on the functions and callbacks provided by the *WASM for Proxies C++ SDK*[8].
- Despite Emscripten supporting C++ exceptions, it is not implemented inside the SDK and not supported by the host implementation (Envoy). It is an active development field, but, at the time of writing, no implementation is yet available[9]. This leads to a twofold problem: it is not possible to program code based on exceptions, but more importantly, libraries already programmed with exceptions will have problems at compilation time.

Therefore, the design has to rely on the C++ SDK realized explicitly for WASM applied on proxies. It offers:

- *Root context object*: With a lifetime equal to the VM instance, it can be used for elements that have to outlive a single stream.

[6] https://emscripten.org/

[7] https://emscripten.org/docs/compiling/Building-Projects.html

[8] https://github.com/proxy-wasm/proxy-wasm-cpp-sdk

[9] https://github.com/proxy-wasm/proxy-wasm-cpp-host/issues/116

- *Context object*: A different instance is generated for each stream and lives up to the end of it. It can be used for all the elements related to a specific request.
- *Callbacks*: As introduced in section , Envoy handles the requests based on events. Therefore, the main logic of the security control will be applied to the callbacks triggered by the events eventually relying on data stored inside the context of the single stream or of the whole instance. The callbacks that can be implemented are divided into two categories:
  - Callbacks for stream events: they are called each time a stream provides certain data, such as headers and bodies. Here has to be implemented the logic that directly interacts with data that have to be checked. The full list of overridable functions is:
    - `onCreate`: Called at the beginning of a stream, the creation refers to the context object.
    - `onRequestHeaders`: Called when request headers are available.
    - `onRequestBody`: Called when the body of the request is decoded.
    - `onRequestTrailers`: Called when request trailers are available.
    - `onResponseHeaders`: Called when response headers are available.
    - `onResponseBody`: Called when the body of the response is decoded.
    - `onResponseTrailers`: Called when response trailers are available.
    - `onDone`: Called at the end of the stream.
    - `onLog`: Called after `onDone` to perform log activities.
    - `onDelete`: Called when there is nothing left to do other than deallocate the flow context.
    - `onTick`: Called when a timer configured via `setTickPeriodMilliseconds` is fired.
  - Callbacks for initialization events: they refer to the VM status. They should be used for performing initialization tasks, load configurations, gracefully ending the security control. The available functions under this category are:
    - `onConfigure`: Called as soon as the WASM module is loaded.
    - `onStart`: Called at the end of the loading, before serving any stream.

8.4 WASM security control development lifecycle

Figure 8.1 shows a schematic overview of all the steps needed to develop and build a WASM module. In the generic example considered, the module depends on libraries not yet imported to WASM architecture, therefore also their build process is presented. The steps to be taken in sequence are:

1. First of all, the source code related to the libraries that have to be ported has to be found and, if needed, fixed (see Section 8.6.1).
2. Via Emscripten (see appendix A.1.1), libraries can be built generating static libraries (`.a` files) compatible with WASM architecture.
3. The core logic of the security control has to be developed.
4. Bazel build (see appendix A.1.2 is in charge of wrapping all the elements together. Specifically, it will build the desired WASM module using the provided libraries as dependencies and relying on the SDK, automatically downloaded by Bazel itself.
5. Once the WASM module (`.wasm` file) is generated, it has to be uploaded to be directly downloadable by an HTTPS request. One of the most immediate solutions is uploading it inside a GitHub repository.
6. Referring to 6.3, a YAML file can be written in which two EnvoyFilter resources are present. The latter will refer to the provided URL to retrieve the WASM module.

7. Finally, Kubernetes ad Istio control plane is exploited propagating the configuration across the whole service mesh. All the Envoy proxies of the selected workloads will receive the WASM module, integrating it inside their HTTP filter chain and enforcing the security control.

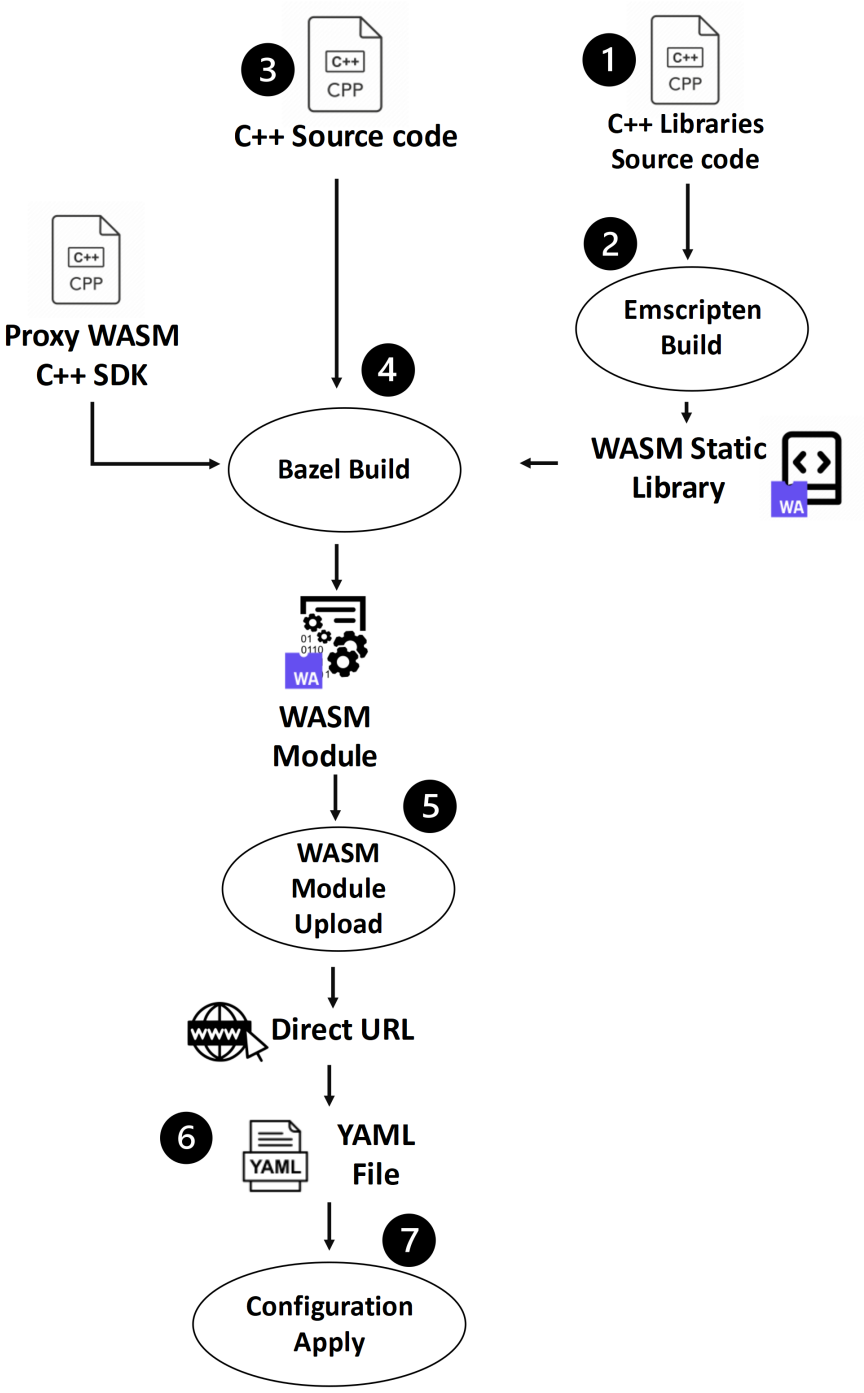

Figure 8.1: WASM module development lifecycle schema

8.5 WAF module: design and implementation

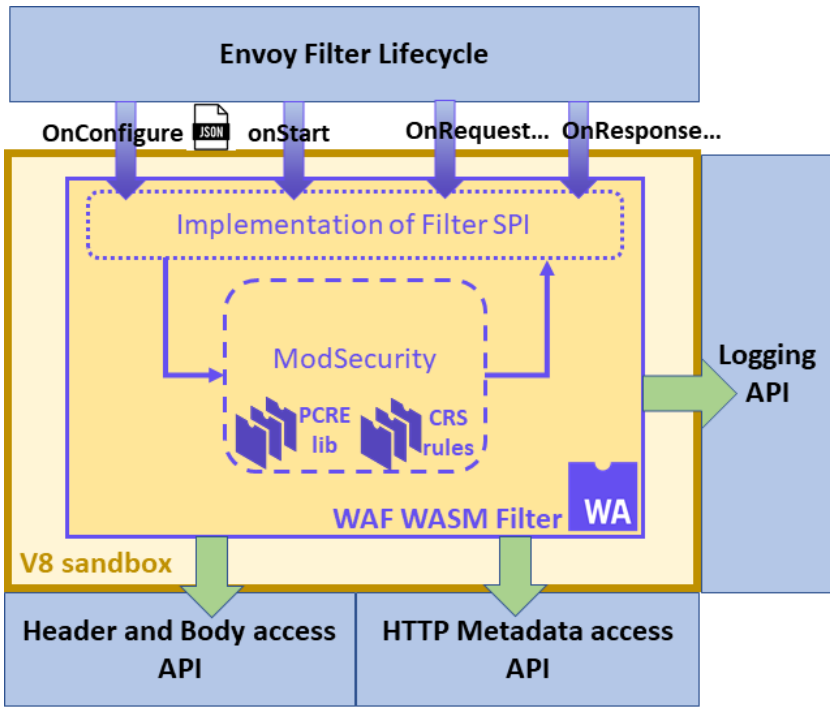

Figure 8.2: WAF Module architecture

Figure 8.2 provides a reviewed and contextualized proxy wasm architecture (see Figure 6.1) for the provided solution. The WASM box has been enriched with ModSecurity's related elements while external entities have been reduced up to just the ones used by the security control in place. Another point of view is provided by figure 8.4 which shows the complete lifecycle of a request that reaches the HCM and goes through the filter chain made of our custom filter and the Envoy router filter. The following subsections will describe in more detail the whole design referring to these two figures.

8.5.1 Configuration

In terms of deploying the filter inside the HTTP filter chain, the configuration is straightforward: as described in section 6.3, all the complexity is handled by the control plane and, just like any other generic filter that may be added, two EnvoyFilter are used. The solution, furthermore, takes advantage of the second EnvoyFitler not only to provide the URI of the WASM module, but also to describe how the WAF will have to behave. It is a key point regarding the flexibility objective: all the configurations are bound inside a single YAML file that may punctually describe a single deployed WAF or scale-up deploying multiple WAFs with the same configurations. A configuration example is shown in the snippet in figure 8.3. Via the `configuration` key, a JSON object is provided, the ABI supports the possibility to retrieve it from the wasm filter during the configuration phase. It is shown in figure 8.2 in which `onConfigure` function comes with a JSON symbol alongside. This function, provided by the Service Provider Interface (SPI), can be, therefore, implemented to program the logic that will consume the configuration and enable ModSecurity accordingly. Table 8.1 describes each JSON parameter, while the complete JSON schema can be found in appendix A.4.

8.5.2 Middleware

Being an event-based architecture, the main logic of the filter resides on the callbacks that are triggered by the events originated by the HCM. The callbacks provided by the SPI may handle

Key	Possible values	Description
enable_default	yes, no	Includes the basic needed rules to turn on ModSecurity and support the CRS
enable_crs	yes, no	Includes all the rules provided by the CRS.
enable_sqli	yes, no	Includes a subset of rules provided by the CRS to intercept SQLi attacks. This rules are already included enabling \code{enable_crs}
enabe_xss	yes, no	Includes a subset of rules provided by the CRS to intercept XSS attacks. This rules are already included enabling \code{enable_crs}
custom	strings	Array of strings to provide custom rules. It is also possible to insert a fully customized configuration explicitly disabling the other keys

Table 8.1: Description of JSON configuration object

```
[...]
configuration:
   '@type': type.googleapis.com/google.protobuf.StringValue
   value: |
     {
       "modsec_config": [
         {
           "enable_default": "yes",
           "enable_crs": "yes",
           "enable_sqli": "yes",
           "enable_xss": "yes",
           "custom_rules":[
           "SecRule ARGS \"@rx matteo\" \"id:103,phase:1,t:lowercase,deny\""
           ]
         }
       ]
     } [...]
```

Figure 8.3: Snippet of EnvoyFilter with WAF configuration

events related to the lifecycle of the WASM module itself or of an incoming stream. Just like the `onConfigure` function anticipated, each one handles a specific moment and interacts with Modsecurity to provide all the needed information that may be checked accordingly to the configured rules. The following are the ones implemented inside the proposed solution:

- `onConfigure`: Called when the WASM module is being loaded by Envoy. This callback parses the JSON object that holds configuration data, instantiate Modsecurity object, setups and provides to it the concatenation of the rules that have to be checked.
- `onRequestHeaders`: Called when request headers are available to be fetched. Basic information of the connection and headers are collected and provided to Modsecurity. Here, as shown in figure 8.4, the WAF process them, and, if any rule matches, a request to interrupt the filter chain is returned back. Once `onRequestHeaders` is completed, the filter chain goes on and the next filter becomes capable of fetching them up to the router filter that forwards the stream to the destination. As later on described in section 8.6, it is an encountered problem during the design phase. There is not a buffering capability that permits handling the headers waiting for completing the request and only at that point forward it.
- `onRequestBody`: Called when the body of the request is ready to be fetched. It will be called only if the request actually has a body: GET requests, if not crafted abnormally, will not trigger this callback. Here the body is added into Modsecurity and processed.
- `onResponseHeaders`: This callback and the next one, being about the response, will be triggered only if the stream has been checked with no detection by the WAF and forwarded

by the router. Then, the response from the application container will cross back the filters triggering this callback as soon the headers are available. Just like the previous ones, Modsecurity will receive them and determine if any action has to be taken.

- `onResponseBody`: The last element taken into consideration by the filter is the body of the response calling Modsecurity's processResponseBody function to analyze it.
- `onDelete`: It decrees the end of the stream and that objects related to it are up for deconstruction. This callback has been implemented to perform memory operations cleaning up the ModSecurity transaction in place.

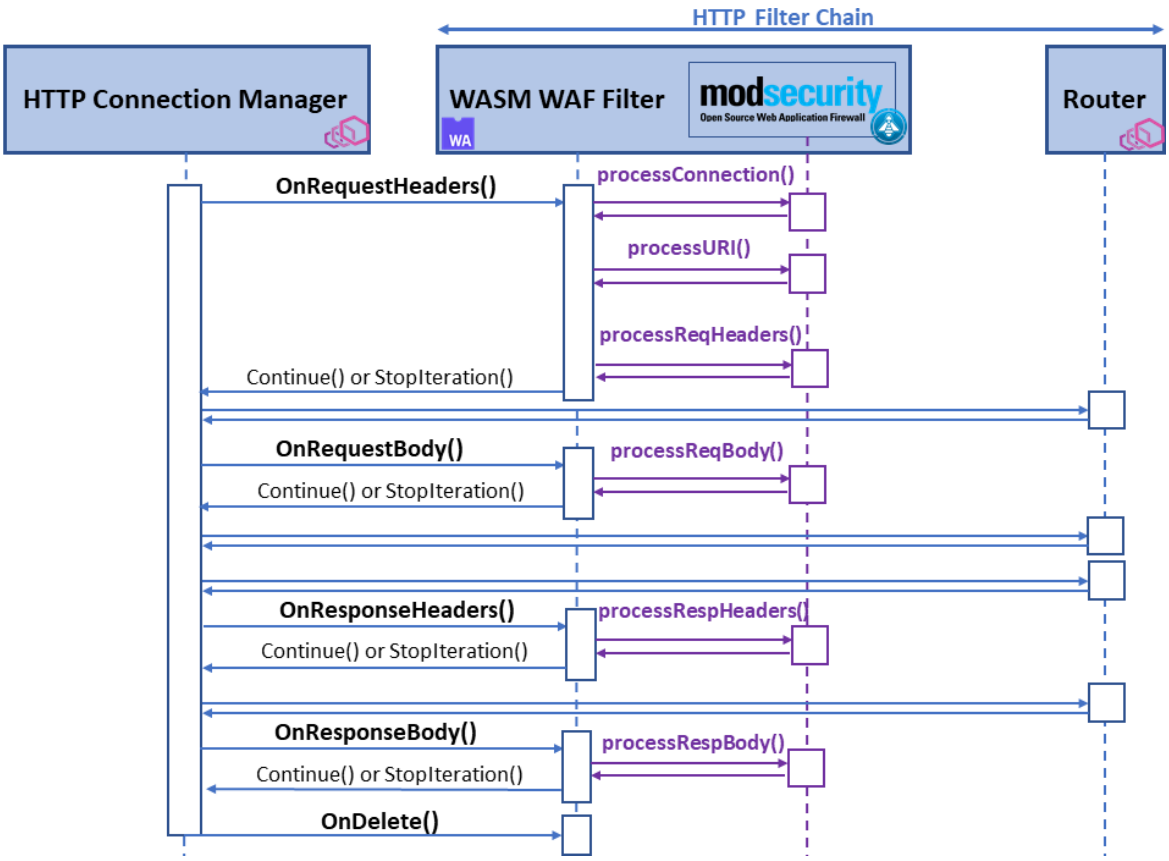

Figure 8.4: Lifecycle of a request interacting with WAF Module

8.5.3 Modsecurity library

ModSecurity integration has been designed to be as light as possible due to the known building problem that would have been raised (see Section 8.3). It includes only the mandatory library that is the *PCRE library*, needed for regular expression pattern matching, and *libinjection*, already embedded inside the code of the library. As described, all the callbacks interact with Modsecurity providing as soon as they are ready all the elements of the stream. As they arrive, a Modsecurity transaction is gradually populated, obtaining increasingly elements to be taken into consideration for the rules. The problem raised about the absence of buffering is mitigated by this transaction that, by design, remains alive for the whole stream and, with the information, it has every time it is processed, takes a decision about letting the stream go on or interrupting it because some rules are triggered. Internally to the filter, the complete CRS, like all the basic Modsecurity configurations, have been hardcoded. It has been done because it is not possible to perform any file system operations (see section 8.3), so the files, usually just read by the Modsecurity, were not usable. It still permits a decent amount of configurability: any hardcoded rule can be disabled via the custom rules writable inside the YAML and any configuration con be overwritten. The

detection mode enabled is the traditional detection mode, where any triggered rule will result in a disruptive action.

8.5.4 Logging

There mainly are two sources of logs: the callbacks to trace the flow of the requests and Modsecurity. Because of the aforehand mentioned problems on execution file system operations, the only log system used is the envoy log. A strict nomenclature has been implemented to help readability and all the callbacks and ModSecurity functions print the evolution of the stream, but, intrinsically, it is a limited solution compared to the high volume of audit logs generated by a standard Modseucity installation.

Figure 8.5 shows the main logs of a complete request handled and logged by the module. A less verbose log has also been designed, at compile time it is possible to disable a `DEBUG` directive reducing the verbosity of the generated module. As the name says, it is mainly for debugging purposes logging each request in its entirety. The elements omitted to disable this directive can be also seen in the snippet of figure 8.5 logging for lines with the `[DEBUG]` tag.

The log snippet in figure 8.6, on the other hand, shows a detected attack. This snippet starts printing the body of the request in which POST arguments are included (line `2`). Despite being URL encrypted, it is already possible to see an attempt to bypass the authentication made via a SQL injection performed on the `passwd` field. The request body is processed (line `5`) resulting in a triggered rule. Line `6` provides all the information about the intervention of ModSecurity: it is possible to see that the matched rule is the one with id `942100` with a message relative to that rule, the specific, decoded, element inside the body that triggered it (`'OR 1=1--'`) and other information such as the rule set to which the rule belongs (CRS) and the paranoia level in place. One last important element resides at the beginning of the line, in which is shown the action that has to be made (return an error code `403`) and the ModSecurity phase in which this rule has been triggered. In this example phase 2 correctly means that the analysis has been performed with the request body provided to ModSecurity (see section 7.2.2). Because of this intervention, the stream ends with a `ResponseHeaders` event (lines `8-11`) in which the HTTP status `403 Forbidden` is returned back to the client.

8.6 Criticalities encountered

This section is intended to group up all the criticalities encountered designing and developing the WASM WAF Filter:

8.6.1 Building Modsecurity and its libraries

As expected (see section 8.3) compiling the library required also to take into account all its dependencies. The design solution to restrict them to just the mandatory one has been a valid choice even if with some impacts in terms of functionalities. The following list provides the excluded libraries and their potential capabilities:

- *GeoIP/MaxMind*: Needed to perform IP-based geolocation and take action if the connections are coming from the so-called "high risk country locations".
- *LibCURL*: Needed to perform remote logging. Given the restricted sandbox on which the module is executed, even adding this library would not be enough to perform remote logging because calls are restricted to the internal cluster and must be performed via the API calls. Further work would be required.
- *YAJL*: Modsecurity relies on YAJL to process JSON and enableJSON format logging.

```
[plugin.cc:405]::initTransaction() [initTransaction] New connection
    10.42.0.1:0 -> 10.42.1.20:9080
[initTransaction] processConnection intervention_ret = 0
[plugin.cc:415]::initTransaction() [initTransaction] Connection setup done
[initTransaction] processURI intervention_ret = 0
[initTransaction] URI information added
[plugin.cc:305]::onRequestHeaders() [onRequestHeaders][DEBUG] id:2
[plugin.cc:306]::onRequestHeaders() [onRequestHeaders][DEBUG] #headers: 25
[plugin.cc:308]::onRequestHeaders() :path -> /login
[plugin.cc:308]::onRequestHeaders() :method -> POST
[...]
[onRequestHeaders] addRequestHeader intervention_ret = 0
[plugin.cc:326]::onRequestHeaders() [onRequestHeaders] Request Headers added
[plugin.cc:330]::onRequestHeaders() [onRequestHeaders][processRequestHeaders]
    correctly executed
[onRequestHeaders] processRequestHeaders intervention_ret = 0
[plugin.cc:369]::onRequestHeaders() [onRequestHeaders] Request Headers
    processed with no detection
[plugin.cc:459]::onRequestBody() [onRequestBody][DEBUG] bodyString =
    username=admin&passwd=adminpwd
[plugin.cc:470]::onRequestBody() [onRequestBody] Request Body added
[plugin.cc:474]::onRequestBody() [onRequestBody][processRequestBody]
    Correctly executed
[onRequestBody] Request Body processed with no detection
[plugin.cc:513]::onResponseHeaders() [onResponseHeaders][DEBUG] Printing
    response headers: 11
[plugin.cc:515]::onResponseHeaders() :status -> 302
[plugin.cc:515]::onResponseHeaders() location ->
    http://istionuc.k3s/productpage
[plugin.cc:515]::onResponseHeaders() set-cookie ->
    session=eyJ1c2VyIjoiYWRtaW4ifQ.YXbROA.fSlsEmLBhSHlzgerivViLEga9Ls;
    HttpOnly; Path=/
[...]
[plugin.cc:528]::onResponseHeaders() [onResponseHeaders][DEBUG] response
    status: 302
[onResponseHeaders] addResponseHeader intervention_ret = 0
[plugin.cc:537]::onResponseHeaders() [onResponseHeaders] Request Headers added
[onResponseHeaders] processResponseHeaders intervention_ret = 0
[plugin.cc:549]::onResponseHeaders() [onResponseHeaders] Request Headers
    processed with no detection
[plugin.cc:573]::onResponseBody() [onResponseBody][DEBUG] responseBodyString =
[...]
[plugin.cc:584]::onResponseBody() [onResponseBody] Response Body added
[plugin.cc:588]::onResponseBody() [onResponseBody] processResponseBody()
    correctly executed
[onResponseBody] Response Body processed with no detection
[plugin.cc:608]::onDelete() [onDelete][OK] 2
```

Figure 8.5: Snippet of logs with a complete request handled

- *LMDB*: Needed to use LMDB (Lightning Memory-Mapped Database) as persistent storage. Its support with libModSecurity, even in more conventional implementation, seems to be

```
1  [...]
2  [plugin.cc:459]::onRequestBody() [onRequestBody][DEBUG] bodyString =
       username=ad&passwd=\%27OR+1\%3D1--+\%27
3  [plugin.cc:470]::onRequestBody() [onRequestBody] Request Body added
4  [...]
5  [plugin.cc:474]::onRequestBody() [onRequestBody][processRequestBody]
       Correctly executed
6  [plugin.cc:151]::process_intervention() [process_intervention] Log: [client
       10.42.0.1] ModSecurity: Access denied with code 403 (phase 2). detected
       SQLi using libinjection. [file "<<reference missing or not informed>>"]
       [line "5419"] [id "942100"] [rev ""] [msg "SQL Injection Attack Detected
       via libinjection"] [data "Matched Data: s&1c found within ARGS:passwd:
       'OR 1=1-- '"] [severity "2"] [ver "OWASP_CRS/3.3.2"] [maturity "0"]
       [accuracy "0"] [tag "application-multi"] [tag "language-multi"] [tag
       "platform-multi"] [tag "attack-sqli"] [tag "paranoia-level/1"] [tag
       "OWASP_CRS"] [tag "capec/1000/152/248/66"] [tag "PCI/6.5.2"] [hostname
       "10.42.1.20"] [uri "/login"] [unique_id "1635177011"] [ref
       "v2368,11"]intervention.status = 403
7  [plugin.cc:166]::process_intervention() [process_intervention] Intervention,
       returning code: 403
8  [plugin.cc:513]::onResponseHeaders() [onResponseHeaders][DEBUG] Printing
       response headers: 3
9  [plugin.cc:515]::onResponseHeaders() :status -> 403
10 [...]
11 [plugin.cc:528]::onResponseHeaders() [onResponseHeaders][DEBUG] response
       status: 403
12 [onResponseHeaders] addResponseHeader intervention_ret = 0
13 [plugin.cc:537]::onResponseHeaders() [onResponseHeaders] Request Headers added
14 [onResponseHeaders] processResponseHeaders intervention_ret = 0
15 [plugin.cc:549]::onResponseHeaders() [onResponseHeaders] Request Headers
       processed with no detection
16 [plugin.cc:608]::onDelete() [onDelete][OK] 10
```

Figure 8.6: Snippet of logs with a detected SQLi attempt

still not stable[10].

- *LibXML2*:Needed to process xml payloads.
- *SSDEEP*: Used to perform fuzzy hash matching to detect web-based malware such as reverse shells scripts and backdoors
- *LUA*: Needed for writing ModSecurity rules based on LUA language.

Despite having been a problem encountered, potentially any of the mentioned libraries may be compiled into WASM and integrated into the module. Another building problem encountered has been the usage of system calls, mainly related to file system operations and exceptions, not suitable for the sandbox environment. About that, a conversation with the Envoy community has raised detecting the functionalities that should be implemented into Envoy project and isolating the problem via a custom version of ModSecurity in which these functionalities have been omitted[11]. It certainly represents another limitation in terms of providing a complete ModSecurity WAF, but it has been a needed tradeoff to find out common ground between ModSecurity requirements and

[10] https://issueexplorer.com/issue/SpiderLabs/ModSecurity/2601

[11] https://github.com/envoyproxy/envoy/issues/17463

the WASM ecosystem capabilities. All of these limitations may be overcome with future releases and extended support.

8.6.2 Buffering

Another common ground to be found was between ModSecurity functionalities and Envoy about having the needed information to let the security control take decisions and be still in time to apply them. Despite being a useful functionality, the possibility of buffering requests is not implemented due to limitations in working at scale in terms of memory limits and traffic size[12]. This limitation leads to the impossibility of waiting for the request body before sending anything the application or modifying the headers at the body phase (because they already have been sent). Despite this, the design provided is based on taking security decisions based on all the elements available at that time and keeping them in memory for the whole stream lifetime. So, partial decisions will be taken, based on them the request will go on or be blocked, while gradually the decision will be based on more information. It may lead to some connection established and then dropped because of elements found afterwards, but still, all the elements sent have been analyzed. One clear example about this is if, analyzing the response, inside the headers, everything is correctly checked, but an information leak is detected inside the body. At the time of the detection, the headers are already sent back to the socket, but then the stream is interrupted. Despite receiving an error, the client will receive a legit header, but an empty body. The security effectiveness of the control is still guaranteed.

8.6.3 Debug capabilities

WASM ecosystem about non-Web embeddings and specifically about the proxy-WASM, still lacks an easily debuggable environment. Envoy log statements with trace-level verbosity have been used. Developing a module that is going to be added into the path that the traffic will take inside Envoy, has to be done with extreme care. Any development decision will have a direct impact on the performance of the system, with the risk of creating a bottleneck. Furthermore, the execution of the WASM module inside the V8 sandbox isolates the execution, but the whole system is still compromised in its functionalities if the module is in an error state, not allowing the filter chain to go on. Envoy keeps working, but the filter chain is interrupted. About it, a test has been performed to analyze the behaviour of the system with a module affected by a memory leak. Once the module execution stops working, the workload becomes unreachable until manual operations are performed removing the WASM Filter from the chain. See section 10.2 for detailed considerations.

8.6.4 WAF Rules: support and effectiveness

Despite the vast majority of CRS rules have been correctly implemented, some of them have been excluded because of optional libraries not implemented (e.g. rule about IP geolocation based on GeoIP) and lack of a way to provide files for rules relying on `.data` files. It would be possible to hardcode them, but libModsecurity does not support an inclusion directly via code. The complete list can be found in appendix A.5.
The effectivenes of some rules depends on the WAF point of application. Internally to a cluster, some rules may receive no relevant connection information. An example is about the IP address: during an interaction between proxy-sidecars, the source IP will always refer to the IP of the pod, not the original external IP.

[12] https://github.com/proxy-wasm/proxy-wasm-cpp-host/issues/143

Chapter 9

Use cases

Based on a sample microservice application, this chapter aims to provide practical examples of zero-trust principles applied via the security capabilities offered by the technological layers presented in this thesis, up to the proposed WAF implemented as a WASM filter and deployed via Istio control plane. The combination of these functionalities leads towards a zero-trust architecture in which defense in depth and least privilege principles are applied. Any legit request is authenticated, authorized and validated. All the other requests, coming from unintended sources or malformed are denied.

9.1 Bookinfo scenario

The scenario used, named *Bookinfo Application*, comes from the Istio repository as an example of a microservice-based application[1]. This application is executed on a small cluster made of two nodes:

- *master node*: Where the control plane is deployed.
- *worker node*: Where users' workloads are running.

It is made of four microservices that, interacting with each other, serve the external client requests coming through the ingress gateway. Specifically, the use case is about providing information about a book retrieving its details, readers' reviews and ratings. These features are reflected in the separations of duties between the deployed microservices:

- `productpage`, developed in Python, is the front-end microservice, the ingress gateway points to this microservice. When a request comes, `productpage` calls the details and reviews microservices to retrieve information that has to be displayed.
- `details`, developed in Ruby, receives requests from `productpage` and gives back book details. A single version of this service is deployed.
- `reviews`, developed in Java, receives requests from `productpage` and provides the readers' feedbacks. Three different versions of the service are deployed and traffic can be split across them thanks to weight policies described via virtual service resources:
  - `v1`: The basic version of the service, just return the textual reviews, without contacting any other microservice to retrieve the ratings.

[1] https://github.com/istio/istio/tree/master/samples/bookinfo

  - `v2` and `v3`: Call the `ratings` microservice to retrieve the numerical value of the rating and return it alongside the textual review. The two versions are graphically distinguished because `v2` returns the rate with black stars while `v3` with red stars.

- `ratings`, developed in node.js, is a single version microservice that provides the ranking information to the `reviews` microservice.

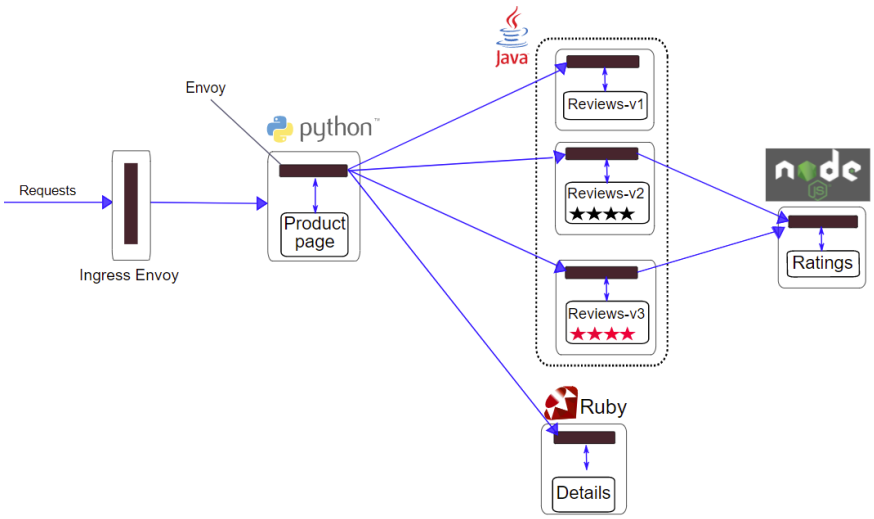

Figure 9.1: Bookinfo application schematic (source: istio.io)

Figure 9.1 provides a representation of the whole application and interaction between microservices. As it is possible to note, the schematic shows the deployment inside the Istio service mesh: all the communications are performed between the dark rectangles representing Envoy proxies. Each application container just interacts with the related proxy inside the pod. A similar representation (shown in figure 9.2) that provides the actual architecture deployed can be also retrieved by Kiali. Here logical boundaries of the applications and namespaces are also shown. In the figure, we can see the namespace `istions` in which the application is deployed. The triangles represent the service nodes linked to the workloads (square elements) that, like the reviews microservice, can be versioned. Kiali representation, compared to the previous schematic, does not provide a static view of the intended deployment, but rather the status of the actual deployment in a selected time span. The coloured arrows show the health of the traffic between each component. They permit the detection of specific points of failure or bottlenecks. Further information about Kiali can be found in Istio observability section 5.5.

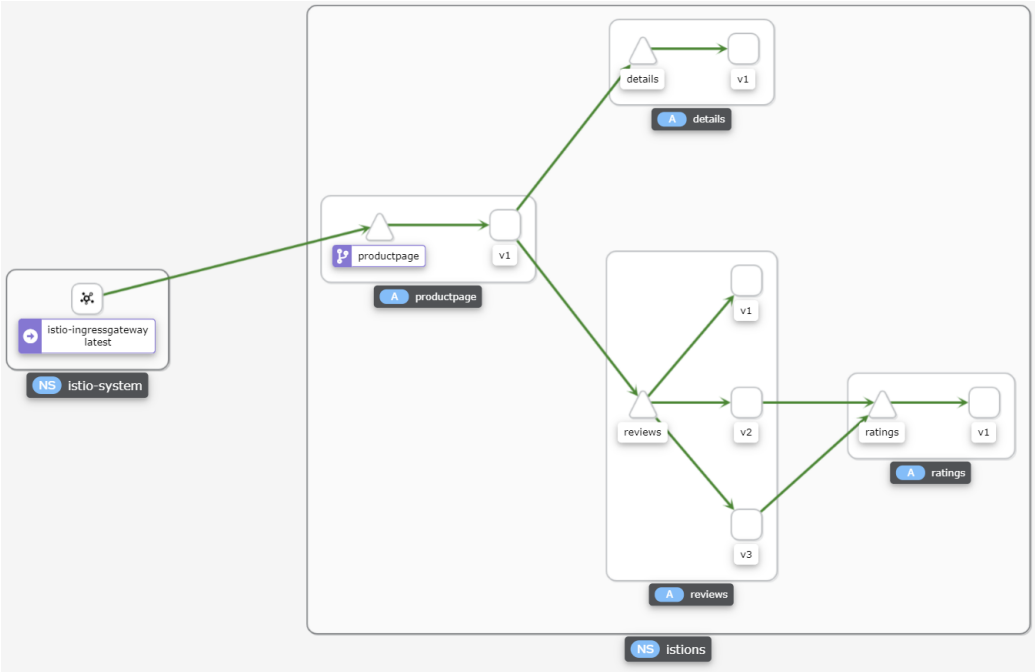

Figure 9.2: Bookinfo application graph provided by Kiali

9.2 Mutual TLS inside the mesh

Providing encryption by default of the traffic inside the mesh is one of the capabilities introduced by Istio (see section 5.4.2) that most affect the security posture of a microservice infrastructure. As zero-trust states, even an internal network has to be considered with threats. Enforcing mTLS provides countermeasures to eavesdropping and man in the middle attacks, and, thanks to the strong identity provided for each workload, permits also to enable authentication and authorization policies, reducing the possibilities of lateral movements originating from any compromised pod. When a connection requires the frontend page, the `productpage` workload receives the request and contacts both `details` and `reviews` to retrieve the needed information. By default, with mTLS set to permissive, intercepting the traffic originated by the `details` workload we can see that the traffic is encrypted. We can do it executing tcpdump on the sidecar container of `details` application:

```
kubectl exec -it details-v1-79f774bdb9-bw2pd -c istio-proxy -n istions --
    sudo tcpdump -i eth0 src port 9080 -A
```

Executing a request via the browser the output is:

```
details-v1-79f774bdb9-bw2pd.9080 >
    10-42-1-68.productpage.istions.svc.cluster.local.49624: Flags [P.], seq
    1:2093, ack 212, win 501, options [nop,nop,TS val 2528710534 ecr
    1438983329], length 2092
E.....@.@.v.
*.J
*.D#x....q.^.!......4.....
....U.$.....N...J..a.j.i.fKZ.A...
[...]
```

We can not see any meaningful information. On the other hand, disabling mTLS for the namespace in which the application is deployed via a `PeerAuthentication` resource:

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
```

```
metadata:
  name: "default"
  namespace: "istions"
spec:
  mtls:
    mode: DISABLE
```

And intercepting again the traffic, a plain request is shown on the screen:

```
details-v1-79f774bdb9-bw2pd.9080 >
    10-42-1-68.productpage.istions.svc.cluster.local.54390: Flags [P.], seq
    1:1216, ack 1437, win 501, options [nop,nop,TS val 2529305225 ecr
    1439578021], length 1215
[...]
HTTP/1.1 200 OK
content-type: application/json
server: istio-envoy
[...]
{"id":0,"author":"William
    Shakespeare","year":1595,"type":"paperback","pages":200,"publisher":"PublisherA",
"language":"English","ISBN-10":"1234567890","ISBN-13":"123-1234567890"}
```

For a graphical representation of the use of mTLS across the service mesh, Kiali provides a security option that displays a closed lock icon on the arrow between two workloads and shows statistics in terms of the percentage of traffic encrypted. Figure 9.3 shows how it is displayed. It also has to be

Figure 9.3: Kiali representation of mTLS traffic

noted that, by default, Envoy proxies are configured with PERMISSIVE mTLS, accepting both encrypted and plaintext traffic. This can be proven by deploying another container even, as a matter of example from another namespace without the sidecar injection enabled, and performing a direct request from the container to the productpage IP.

```
kubectl run curl-pod --image=radial/busyboxplus:curl -i --tty --rm -n default
curl 10.42.1.79:9080
```

As expected, a 200 response is returned with the whole page:

```
curl 10.42.1.79:9080
<!DOCTYPE html>
<html>
  <head>
    <title>Simple Bookstore App</title>
[...]
```

On the other hand, by enforcing STRICT mTLS:

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"
  namespace: "istions"
spec:
```

```
  mtls:
    mode: STRICT
```

The same request is now refused. From now on, only authenticated workloads that perform the request providing a valid certificate will be served.

9.3 Authorization of mesh traffic

Providing and verifying identities enables to define what each entity is actually capable of doing. Authorization policies inside the service mesh permit to follow the least privilege principle, restricting each entity to perform only the intended and demanded by design actions and not allowing anyone with a valid certificate to perform any wanted actions. In the bookinfo scenario, as a matter of example, the `productpage` workload does not have to directly interact with `ratings` workload, but rather information about the ratings should be only accessed by `reviews`, the latter workload, later on, will provide the aggregated data of the book. Despite this intended behaviour, direct communication can be manually performed and the details about ratings are retrieved:

```
{"id":0,"ratings":{"Reviewer1":5,"Reviewer2":4}}
```

Based on the order of rules shown in section 5.4.4, by making explicit the allowed workloads, all the requests coming from others will be denied. It can be done by applying the following AuthorizationPolicy, in which we set an ALLOW policy for the `ratings` workload requiring to accept only GET requests from `reviews` workloads.

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "ratings-viewer"
  namespace: istions
spec:
  action: ALLOW
  selector:
    matchLabels:
      app: ratings
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/istions/sa/bookinfo-reviews"]
    to:
    - operation:
        methods: ["GET"]
```

Performing again a request from `productpage` to `ratings`, now a forbidden error is returned.

```
HTTP/1.1 403 Forbidden
RBAC: access denied
```

The introduction of this authorization policy still does not affect the intended behaviour and the product page is correctly displayed with both reviews and ratings. To achieve an overall better security posture, it is best practice to use a default-deny pattern and, on top of it, granularly authorize the minimum interaction between services to accomplish the required tasks. Allow nothing can be set by an authorization policy namespace wide:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-nothing
  namespace: istions
spec:
  {}
```

Further considerations can still be done. The point of application of the policies is Envoy, the proxy sidecar of the main application container. Traffic generated by the container and not intercepted by the sidecar will not have the allow-nothing policy, and all the other policies, enforced. A description of the supported and intercepted traffic has been made in section 5.6. The following example proves it. Two pods ubuntu-based are deployed inside the `istions`, in which strict mTLS and deny all policy are applied. One pod will act as the server, installing netcat and listening for UDP traffic.

```
kubectl run nc-pod --rm -i --tty --image ubuntu -n istions -- /bin/bash
apt update && apt install netcat
netcat -ul 9001
```

The other one will act as the client, trying to contact the sever.

```
kubectl run nc-pod2 --rm -i --tty --image ubuntu -n istions -- /bin/bash
apt update && apt install netcat
netcat -ul 9002
```

UDP traffic is correctly sent to the destination, effectively bypassing the sidecar and the service mesh policies. To deny it, further defense in depth has to be applied. One simple solution is to add a native Kubernetes NetworkPolicy allowing only TCP traffic:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-no-tcp
  namespace: istions
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - ports:
    - protocol: TCP
```

Contrary to the others, this policy is applied by Kubernetes, not at the service mesh level. So, it is enforced directly by the CNI and is applied to any connection inside the namespace. Doing so, all the traffic that is not TCP is denied while the TCP traffic, that is correctly handled by Istio, is forwarded to the sidecar. The latter will manage it by applying mTLS, fine graned policies and all the functionalities provided by the service mesh. Even adding this network policy, the functionality of the bookinfo application is still not altered being based on HTTP connections. It shows how the defence in depth approach permits to enforce policies on multiple layers reducing even further unexpected traffic inside the cluster, but still, with the proper configuration, permits to fully control the network and all its functionalities.

9.4 End user authentication via JWT

The previous section showed use cases of authentication and authorization for traffic inside the mesh, between services. Taking into consideration the connections from outside, end users are typically not provided with a client certificate that may be used for mTLS. As introduced in section 5.4.3, JSON Web Token are therefore used. In this use case, based on JWT authentication we provide access to the ingress gateway of our cluster. Access will be denied to requests with invalid or missing tokens. To accomplish this task, a *RequestAuthentication* and an *AuthorizationPolicy* are required. The first validates the token based on the public key set provided while, the latter, denies any request without a token. The example is based on a test token and JWKS endpoint provided by the Istio repository[2].

[2] https://github.com/istio/istio/tree/master/security/tools/jwt/samples

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: "ingress-jwt "
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  jwtRules:
  - issuer: "testing@secure.istio.io"
    jwksUri: https://raw.githubusercontent.com/istio/istio/release-1.10/
        security/tools/jwt/samples/jwks.json
----
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "frontend-ingress"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  action: DENY
  rules:
  - from:
    - source:
        notRequestPrincipals: ["*"]
```

Applying them both requests without tokens and invalid ones return `403 Forbidden`:

```
curl -I http://istionuc.k3s/productpage
curl -H "Authorization: Bearer randomToken" -I http://istionuc.k3s/productpage
```

While providing a valid token, the product page front end is correctly shown with `200 OK`:

```
curl -H "Authorization: Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6IkRIRUg[...]" -I
    http://istionuc.k3s/productpage
```

9.5 Securing workloads with WASM WAF Filter

Taking into consideration legit, authenticated and authorized traffic, the content of the traffic itself has still to be considered untrusted: by definition, any element coming from external requests that the user could have been capable of modifying can potentially be malicious. Even considering pods not directly connected to the ingress gateway, some elements of the request may still have been forwarded from the original one. Furthermore, following the principle of zero trust that assumes internal compromisations an authenticated and authorized pod could still start behaving maliciously: performing abnormal requests to retrieve data or trying to escalate the cluster performing lateral movements to compromise other pods, potentially with more privileges. Adding web application firewalls inside this scenario provides another layer of security inside a typical cluster made of web applications highly coupled to interact together. The deployment is flexible enough to be applied to various scenarios and needs. As a matter of example, the following snippet shows the deployment of the WAF directly inside the ingress gateway. Being an Envoy proxy inside the service mesh, it also is a valid position of deployment. In terms of deployment configuration, it just has to set the `GATEWAY` context instead of `SIDECAR_INBOUND`.

```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: EnvoyFilter
metadata:
  name: modsec-productpage
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      istio: ingressgateway
  configPatches:
  - applyTo: HTTP_FILTER
    match:
      context: GATEWAY
     [...]
              runtime: envoy.wasm.runtime.v8
              code:
                remote:
                  http_uri:
                     uri:
                         https://github.com/M4tteoP/wasm-repo/raw/main/basemodsec.wasm
                  [...]
                  "modsec_config": [
                    {
                      "enable_default": "yes",
                      "enable_crs": "yes",
                      "enable_sqli": "yes",
                      "enable_xss": "yes",
                      "custom_rules":[
                      "SecRuleRemoveById 920280"
                      }
                      [...]
```

All the requests handled by the ingress gateway are now checked regardless of the destination workload:

```
curl -I http://kialinuc.k3s/kiali/console/overview?duration=%3Cscript%3E
HTTP/1.1 403 Forbidden
curl -I http://istionuc.k3s/productpage?arg=../../../../etc/passwd
HTTP/1.1 403 Forbidden
```

A more granular deployment example requires having knowledge of the applications used behind each service. For instance, the `ratings` service may rely on a SQL database to collect users' ratings. Therefore, we can deploy a WAF for this service just enabling protection for specific technological stacks. The following configuration just enables CRS rules to prevent SQLi:

```
"modsec_config": [
 {
   "enable_default": "yes",
   "enable_crs": "no",
   "enable_sqli": "yes",
   "enable_xss": "no",
   "custom_rules":[
   ]
```

Now, performing a request to the frontend page (productpage) with a SQLi payload inside the headers (in this request it has been appended to the `User-Agent`) the page is returned, but displaying an error on retrieving the ratings:

```
Ratings service is currently unavailable
```

It means that the malicious payload, has been propagated internally to the cluster even inside the requests performed between `productpage` and `reviews` and `reviews` and `ratings`. On the latter, the WAF denied the request:

```
warning envoy wasm wasm log vm_prodpg: [plugin.cc:308]::onRequestHeaders()
    user-agent -> Mozilla/5.0 (Windows NT 10.0; Win64; x64)
    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.61
    Safari/537.36admin'or 1=1 or ''='
[...]
wasm log vm_prodpg: [plugin.cc:151]::process_intervention()
    [process_intervention] Log: [client 10.42.1.94] ModSecurity: Access
    denied with code 403 (phase 2). detected SQLi using libinjection.
```

The latter example does not want to highlight a possible SQLI injection via `user-agent` parameter, but rather that user inputs can easily propagate inside the cluster even on internal microservices. Zero trust principles can be enforced granularly deploying the WAFs across the service mesh and fine-tuning them to deal with threats that may affect specific workloads.

Chapter 10

Test and Validation

Tests have been performed in order to ensure the functionality of the developed WAF module and to make considerations about its impact in terms of setup time, latency, scalability and CPU overhead. It has been deployed inside a Kubernetes cluster with Istio and a sample application. Given the limited amount of hardware resources and the dummy scenario, tests are not intended to provide rigorous benchmarking. As indicated in Istio documentation[1], performance tests would require a way bigger cluster and the software deployed in the cluster has a big impact. It is still possible to have a gauge about the overhead introduced by each component considering the presented baselines of the defined testbed. Further considerations have to be performed in all the other environments in which this WAF solution is implemented. All the tests have been performed on a testbed made of:

- 2 Intel NUC: Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz with 16GB of RAM, that formed the cluster made of one master and one worker node. They were running:
  - OS: Ubuntu 20.04.3 LTS
  - Kubernetes 1.21: Via K3s lightweight distribution[2].
  - Istio 1.10.1: For both control plane and dataplane.
- Dell Inspiron: Intel(R) Core(TM) i7-11800H @ 2.30GHz with 16GB of RAM, which formed the client. It has been connected to the cluster via a 10/100 Mbps Ethernet switch. It was running:
  - OS: Windows 10, WSL Ubuntu 20.04.2 LTS
  - curl v.7.68.0: Used to perform single functionality tests.
  - httpit v.0.4.0: Used to generate traffic. Httpit is an HTTP benchmark tool written in Go, designed for high performance being based on the fasthttp package[3]. the main tool parameters used to perform these tests are:
    - *Concurrent connections*: Parallel connections originated by the client. Flag: `-c`.
    - *Number of requests*: Total number of requests performed by all the concurrent connections. Flag `-n`.
    - *Requests per second*: Overall number of requests send each second. Flag `-qps`.

The microservice-based scenario used to perform tests is Bookinfo, the one introduced in the use cases chapter, in section 9.1. A typical deployment consists of filters applied in the sidecar of `productpage` pod and with all the pods executed with a single replica. The description of each test will still provide the precise status of the cluster in terms of deployment and Istio configuration.

[1] https://github.com/istio/tools/tree/master/perf/istio-install

[2] https://k3s.io/

[3] https://github.com/gonetx/httpit

10.1 Setup time

Setup times of different versions and configurations of WASM filters have been calculated to see how much time Envoy proxy takes to enable the required custom logic. Times have been taken by Envoy logs with wasm logs set to the most verbose option (`trace` level). The calculated time refers to the seconds between the activation of the WasmVm and the return of `proxy_on_configure`. Both events are logged inside the Envoy logs. These tests have been conducted with Istio installed with default profile and adding the filter to the `productpage` workload, deployed as a single replica. Figure 10.1 provides a comparison between the simplest filter that can be implemented and two versions of the WAF module.

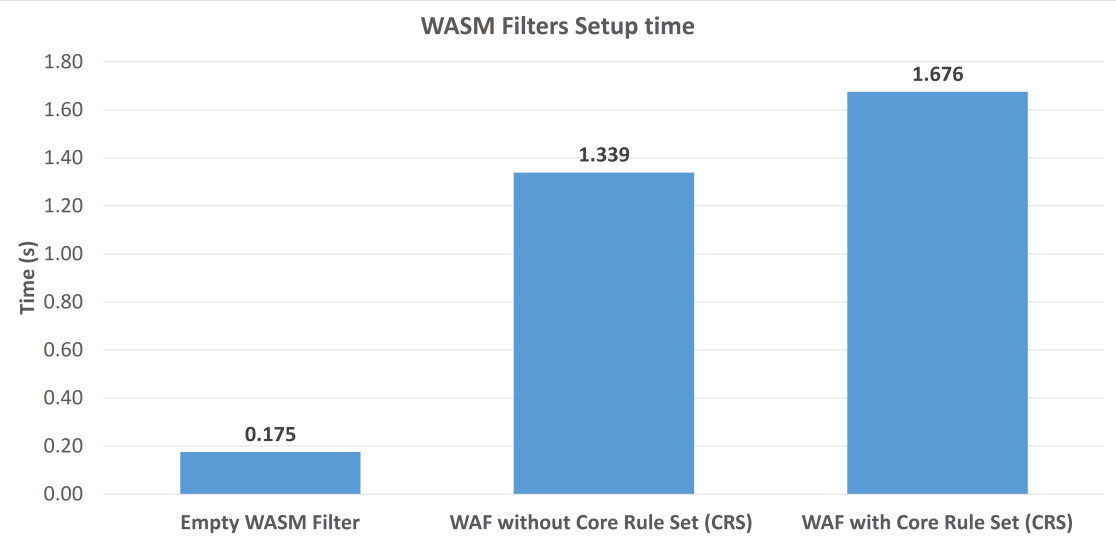

Figure 10.1: Setup time comparison

Empty WASM Filter is an elementary WASM filter in which a minimal forwarding logic is implemented: the traffic just flows through it. It has been developed and tested to provide a baseline of the time needed to load a WASM filter. The difference in loading a more complex module (the proposed WAF) is significant, with an average increase of over one second. Differences in WAF configurations also affect loading time: requiring the complete CRS (*WAF with Core Rule Set*) takes approximately 1.676 seconds, a setup time increase of about 25% compared to not including it. An increase was expected, rules are already hardcoded inside the application but still has to be loaded inside the just born Modsecurity object. Furthermore, the whole rules applied are logged into the Envoy log stream. These numbers can be compared to the startup time of the whole envoy filter that resulted, in the testbed environment being on average 1.663 seconds. It still has to be noted that the addition of the WASM filter is performed at runtime, when Envoy itself is already running.
Another setup test has been performed about the addition of custom rules from the configuration file. Figure 10.2 shows the increment of time adding more rules. Results are comparable with the previous tests in which the hardcoded rules of the CRS have been enabled inside the WAF. This test also showed a current potential limitation in terms of the number of custom rules that can be applied. The YAML field used to provide rules presents a length limitation equal to 262144 bytes. It may be reached with about 3000 simple custom rules.

10.2 Memory leak behaviour

Experiencing memory leak problems during the WAF development, lead to further investigation of the resiliency of the architecture in case of a memory leakage programming error. To perform this test, a WAF module affected by memory leak has been developed. Specifically, the module did not deallocate the Modsecurity object. It is the object in charge of storing the header, the body and

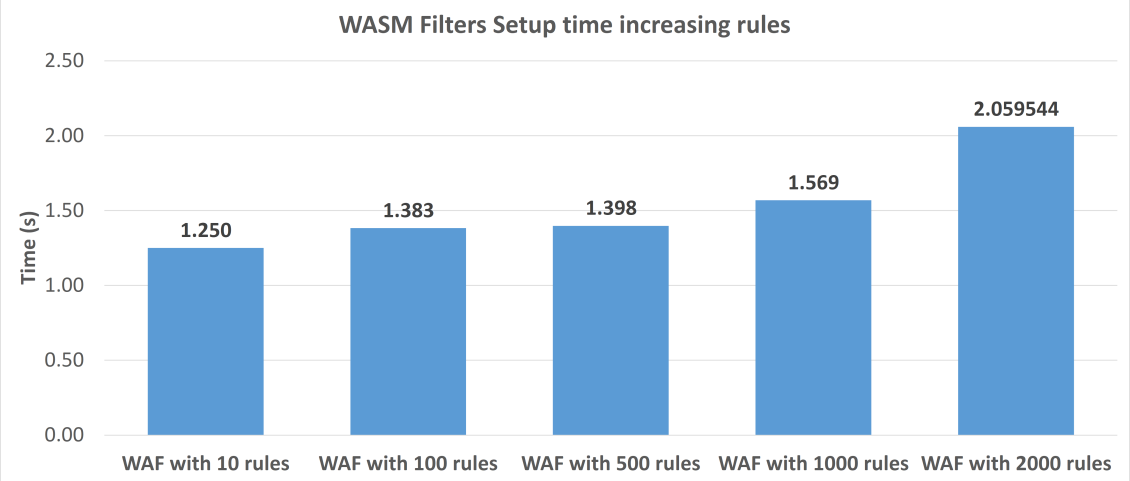

Figure 10.2: Setup time comparison increasing custom ModSecurity rules

all the related Modsecurity information about the specific stream. It is created at the beginning of each stream and, in proper implementation, deallocated once the response is analyzed and sent back to the client. So, one Modsecurity object was leaked for each stream served. With Istio installed as default profile, the filters have been added to the `productpage` workload, deployed as a single replica. It has been then stressed with httpit requiring 100 requests/s via 50 concurrent connections. Figure 10.3 shows the difference between the application's expected behaviour and the WAF affected by a memory leak. 50 concurrent connections rapidly lead the WASM module to an error state. After about 35 seconds, the WAF is no more able to handle any request. In total 1060 requests were correctly returned with a 200 status code.

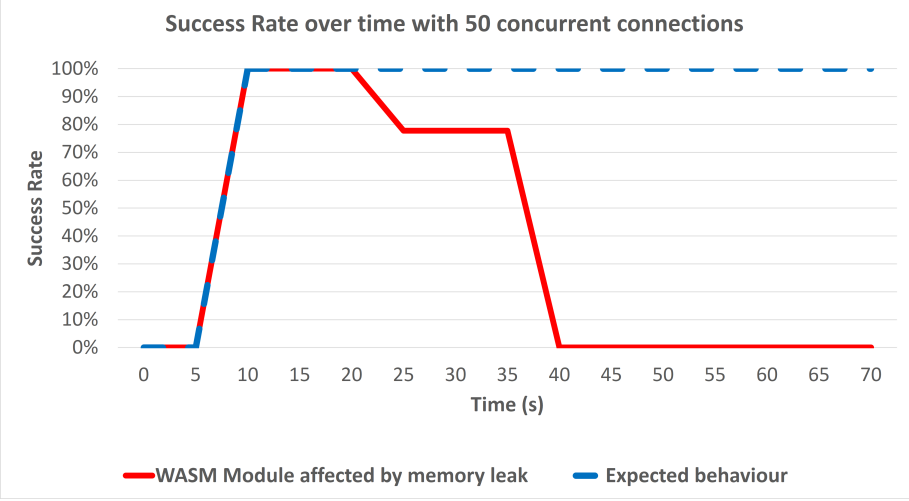

Figure 10.3: Success rate evolution over time with a WAF affected by memory leakage

Once reached, it is a fatal error state described inside the Envoy logs with the following line:

```
error envoy wasm Function: proxy_on_request_headers failed: Uncaught
    RuntimeError: unreachable
```

`proxy_on_request_headers` is a function defined inside the ABI specification. As the name states, it is executed when the HTTP request headers are received. It results to be unreachable, so the whole process is interrupted. Furthermore, the WASM Filter does not return back how the filter chain has to continue, leading to a broken filter chain and to an unaccessible service error (503 Service Unavailable status code) returned to the client. All further requests are logged inside the Envoy sidecar as follows, with the `wasm_fail_stream` label:

```
"GET /productpage HTTP/1.1" 0 - wasm_fail_stream - "-" 0 0 0 - "10.42.0.1"
    "httpit/0.4.0" "09e445bc-a815-99ac-b0ed-2a1ca38ab150" "istionuc.k3s" "-"
    inbound|9080|| - 10.42.1.41:9080 10.42.0.1:0
    outbound_.9080_._.productpage.istions.svc.cluster.local -
```

From the point of view of the user, en error body shows the error returned by the server:

```
upstream connect error or disconnect/reset before headers. reset reason:
    connection termination
```

Further error status can be retrieved running the diagnostic command `dmesg` directly from inside the proxy container:

```
envoy[3495784]: segfault at 0 ip 00005603c53a1e78 sp 00007fecd9202740 error 4
    in envoy[5603c36a6000+2797000]
```

Removing the WASM filter from the chain, Envoy comes back to properly work. So, the initial WASM considerations about executing it inside a sandbox to prevent any other process from being affected may be legit, but still, a broken WASM module deployed inside the chain leads to severe results. To mitigate this behaviour, it would be possible to apply Kubernetes native health checks in charge of rebooting the pod even if all the new ones will always have the sidecar with a deterministic end of life. It is therefore extremely important to correctly develop the filters, being aware of the impacts once they are added to the chain.

10.3 Stability tests

Stability tests have been performed in order to provide feedback about the stability of a microservice application in which a WASM module is deployed. Just like previous tests, the WASM modules have been applied to the `productpage` workload, deployed as a single replica. Three different WASM modules have been tested and all behaved in the same manner:

- *Empty WASM Filter*: Already described in section 10.1, the Empty WASM Filter does not perform any meaningful action on the traffic. The filter is just added to the filter chain and let the traffic flows not reading nor writing it.
- *Basic Auth WASM Filter*: Described in related works (see section 8.1.2), this filter performs a check on the request headers to authenticate the request. It has been used an already provided WASM release[4].
- *WASM WAF Filter*: the developed WAF module.

It has been noted that, stressing over a certain degree of requests/second and concurrent connections, the application starts acting in an unstable way not serving the requests and returning `503` errors. This instability can lead to the same fatal state described in the previous section: from that point, the WASM module is no more capable of handling any requests, returning only error codes and logging a generic `wasm_fail_stream` error.
Figure 10.4 shows two examples of the described behaviour. The left figure shows the success rate over time of requests to `productpage` with the empty wasm filter deployed. With 1000 concurrent connections and unbounded maximum requests per second, the success rate drops to unstable ranges going even below the 50% threshold. The right figure shows the fatal state reached by `productpage` pod with the basic auth filter deployed. In that case, 400 concurrent connections were requiring an unbounded number of requests per second. On the one hand, these results show general instability problems of the WASM for proxies technology in handling concurrent connections. This can be attributed to the alpha phase of the extensibility feature. Further tests

[4] https://github.com/istio-ecosystem/wasm-extensions/releases

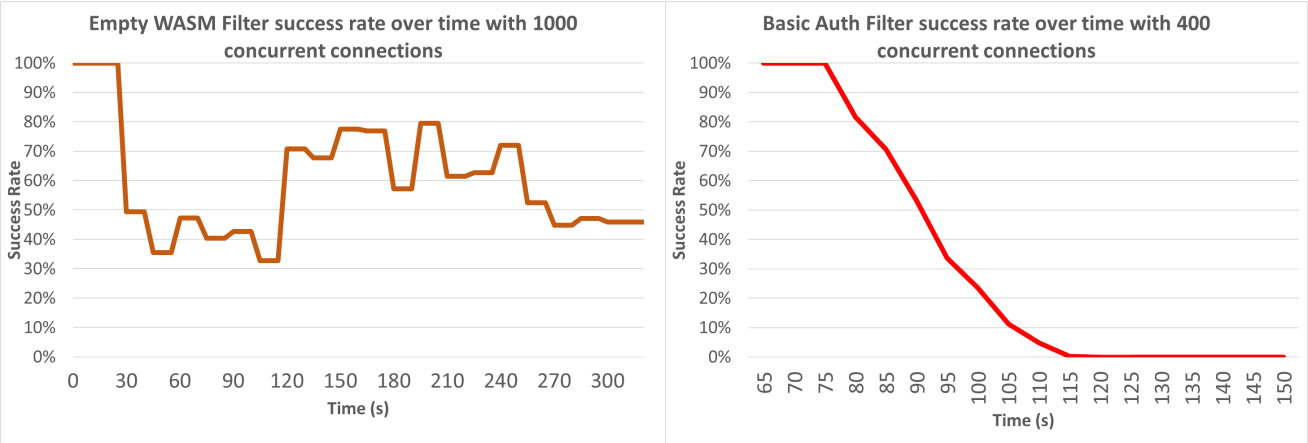

Figure 10.4: WASM Filters stability tests

should be performed for high availability environments before taking into consideration the usage of WebAssembly based modules for production workloads. On the other hand, tests made not exceed the number of requests per second that the application can handle and with an appropriate number of concurrent requests resulted in great results. A one replica product page with the WAF module deployed handled 1 million requests and kept serving them after over 8 hours of tests. Furthermore, being in a distributed environment designed to dynamically scale horizontally, it has been tested that adding replicas permits to split the requests and the concurrent connections to multiple envoy proxies. It leads to higher traffic limits that the application can handle. Further considerations about scalability are presented in section 10.5. It also has to be noted that WASM modules scalability is effortless and automatically handled by the Istio control plane.

10.4 Scalability tests

Scalability tests have been performed scaling the `productpage` pod replicas up to 5 and monitoring the requests that the application is able to handle. Concurrent connections have been fixed to 100. The same test has been executed deploying Istio with the default profile, the Empty WASM Filter and the WAF filter with the CRS enabled. Figure 10.5 shows the results. Adding replicas leads to an increment of the requests per second handled in all the scenarios. This increment becomes less significant from 3 to 5 replicas. It may be mainly due to performance limits introduced by the other microservices that productpage calls (refer to figure 9.2 to see all communications between pods). All of them have been deployed with a single replica. Another result highlighted by the figure is about the overhead introduced by WASM modules. While with the empty wasm the requests handled are similar to the Istio baseline, the WAF modules introduce an evident overhead. It is expected given the complexity of a web application firewall, also considering the whole CRS applied. Further considerations about overhead are shown in the following section, in which different deployments' latencies are compared.

10.5 Latency tests

10.5.1 Latency overhead

Latency tests have been executed with 50 concurrent connections performing a total of 20000 connections originated from the client machine, located outside the cluster. Requests, therefore, had to reach the cluster via the switch and go through the ingress gateway. Figure 10.6 provides a comparison between different deployments: starting from the bookinfo application deployed just on Kubernetes and adding progressively different layers and security features. Note that the baseline of the graph is not 0ms to provide better visualisation of the overhead comparison. As a

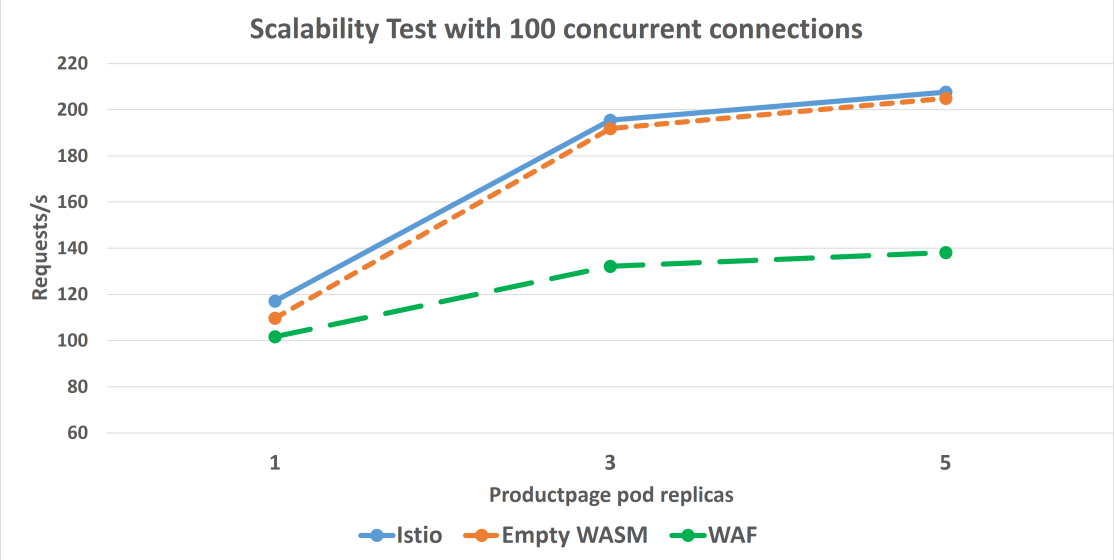

Figure 10.5: Scalability test with 100 concurrent connections

general consideration, it is expected to see a latency overhead adding extra logic applied onto the traffic. Rigorous considerations about it are explained by Istio documentation itself[5]: Istio itself deploying the Envoy proxies and any filter implemented inside, adds the length of the connection path, impacting the latency.

As expected, plain Kubernetes deployment results to be the one with better latency times. The

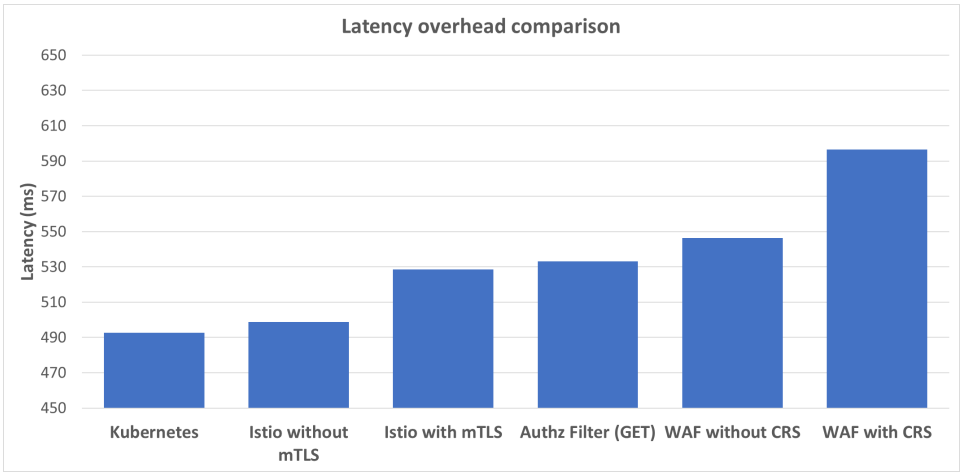

Figure 10.6: Latency overhead comparison

integration of Istio leads to an overhead of about 1.25%, an average of 6ms added. It is higher compared to the official Istio values (claiming an increment of 2.91 ms 99th percentile latency), but, as stated at the beginning of the chapter, a discrepancy between this test with more rigorous one is expected taking into consideration many elements such as the physical network, the bandwidth, the size of the cluster and relative service mesh. The overhead introduced by enabling mTLS inside the namespace of the use case is equal to about 6%. The last three elements shown are about filters. The authorization filter has been tested to provide a comparison between a built-in envoy filter with basic logic (it just checks if the request method is GET, otherwise rejects the request) and a WASM filter with a way complex logic: the WAF developed. The overhead introduced by the latter is about 3 times the overhead of the authorization filter, with an overall overhead

[5] https://istio.io/latest/docs/ops/deployment/performance-and-scalability/

of about 18ms. Compared to the basic deployment of Istio, it is a percentage overhead equal to 3.35%. A not negligible overhead is also introduced by enabling the CRS with the percentage overhead that now reaches 9.22%. An interesting comparison is also between plain Kubernetes and Istio with its default features (such as metrics), mTLS and the WAF deployed. Overall a 10.89% overhead is introduced that reaches 21.11% using the CRS.

10.5.2 Malicious pattern detection overhead

This test has been performed to check what happens if a rule is triggered in terms of latency. The latency dropped from an average of 596.62ms for a successful request, to an average of 93.99ms. It means that triggering the rule does not have a complex logic that introduces significant overhead and, at the same time, dropping the request, the response is sent back way faster: no connections are performed between the first service and all the other microservices to compute the response. Furthermore, the WAF itself as soon as triggers a rule reacts interrupting the filter chain and calling the API to send back the error response. A considerable latency reduction has been a predictable outcome.

10.6 CPU overhead tests

CPU usage has been collected via Grafana, capturing the proxy resource usage with the bookinfo application deployed. All the workloads are executed with one replica, with the exception of `productpage` workload scaled from 1 replica up to 5. Based on tests made with 100 concurrent connections and a stable number of 160 requests per second, figure 10.7 shows the initial CPU overhead between the plain Istio and Istio with the WAF in front of the `productpage` pod. As expected, being the WAF capabilities CPU-intensive, there is a not negligible overhead of about 85% of vCPU usage. Afterwards, the `productpage` pod is scaled reaching 3 and 5 replicas. The increment in CPU usage is not linear with the increment of replicas. Five replicas compared to one lead to an increment of about 58%. This is because each pod had no constraints in terms of vertical scalability so, even just one replica had to require the needed amount of computational power to handle the request. Increasing the number of replicas, the deployment is horizontally scaled. Each replica is less stressed, requiring individually fewer resources and guaranteeing better stability.

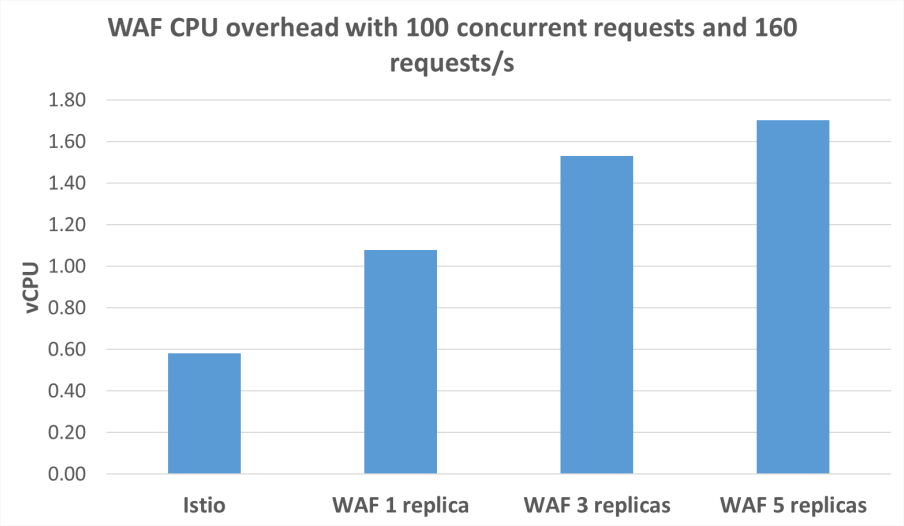

Figure 10.7: CPU overhead tests increasing replicas numbers

10.7 Functional testing

Functional testing has been made in order to verify that the developed WASM module actually works as designed (see chapter 8). Being a research project and given the early phase of WASM technology applied as proxy extensions, it is not intended to be a production-ready security control, but still, it is intended to enrich the security posture against application attacks.

10.7.1 Filter chain

Two HTTP Filters, including one WAF WASM, have been added to the filter chain to analyze if any problem arises. They are added in the proxy of `productpage` workload, in the default bookinfo test scenario. Figure 10.8 shows the status of the chain inside the Envoy proxy.

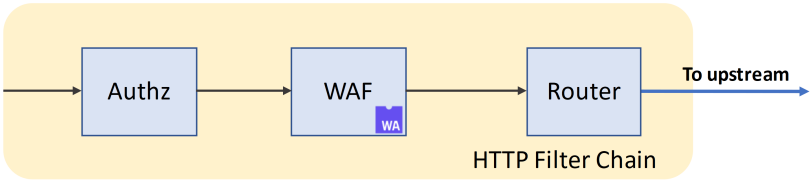

Figure 10.8: Complete set of HTTP filters deployed

Following the incoming traffic flow, the order of the filters is:

- *Authorization Filter*: Deployed via the AuthorizationPolicy CRD of Istio, it is configured to allow only GET methods.
- *WAF WASM Filter*: The complete implementation of the proposed solution with CRS enabled.
- *Router Filter*: Already introduced in section 8.3, it ends the HTTP filter chain.

Sending a POST request via curl:

```
curl -X POST -d 'username=user1' -d 'passwd=password'
    http://istionuc.k3s/login
```

The error message `HTTP/1.1 403 Forbidden, RBAC: access denied` is returned. It is the message provided by the AuthorizationPolicy. As expected, no traffic is logged by the WASM Module because the filter chain has been interrupted before it.
The same outcome happens sending a POST request in which a malicious XSS payload is added:

```
curl -X POST -d 'username=<script>alert('0')</script>' -d 'passwd=password'
    http://istionuc.k3s/login
```

This proves that the chain is actually the one described, the WAF has not detected the threat as the malicious request has been interrupted before being able to analyze it.
Performing a GET request with a malicious payload:

```
curl -v http://istionuc.k3s/login?arg=\<script\>alert\('0'\)\</script\>
```

The return code is `HTTP/1.1 403 Forbidden` without any message. This is the behaviour of the WAF filter: as expected the request goes through the chain and is accepted by the Authz filter. It then reaches the WAF that analyzes and drop it detecting an XSS attack. This is the first action logged by the WASM module. Everything worked as expected, with the WASM filter behaving just like any built-in filter inside the HTTP filter chain.

10.7.2 Effectiveness of WAF security

Security effectiveness tests have been performed to verify the ModSecurity integration inside the WASM module. It has been therefore tests if the module is capable of detecting a wide range of attacks. To do it, *GoTestWAF* tool[6] has been used. It is a tool for attack simulation designed to evaluate security controls such as WAF, IPS and API gateways. Providing an URL target, the tool generates a set of requests with malicious content placed in different parts of the HTTP request (such as inside the headers, body and parameters). Different types of encoding (Base64,URL,XML Entity,Plain) are also performed to further test the security control. Sending these requests, GoTestWAF performs tests against several types of attacks. Table 10.1 shows the whole list of attacks performed and each partial result.

Test set	Test case	Percentage, %	Blocked	Bypassed
community	lfi	66.67	4	2
community	rce	57.14	24	18
community	sqli	87.5	42	6
community	xss	96.71	294	10
community	xxe	100	2	0
owasp	crlf	0	0	0
owasp	ldap-injection	62.5	5	3
owasp	mail-injection	0	0	6
owasp	nosql-injection	50	6	6
owasp	path-traversal	59.09	26	18
owasp	rce	66.67	8	4
owasp	rce-urlparam	33.33	1	2
owasp	shell-injection	66.67	16	8
owasp	sql-injection	50	18	18
owasp	ss-include	50	5	5
owasp	sst-injection	25	4	12
owasp	xml-injection	100	13	0
owasp	xss-scripting	44.12	15	19
owasp-api	graphql	0	0	2
owasp-api	graphql-post	100	3	0
owasp-api	grpc	0	0	0
owasp-api	rest	100	2	0
owasp-api	soap	100	2	0

Table 10.1: WAF WASM module security effectiveness test

Figure 10.9 shows a high-level overview of the outcome of the test. A total of 490 malicious requests blocked (78% of the total), testify that ModSecurity and the embedded CRS actually works. The suboptimal results can be justified considering different aspects: first of all, it was predictable that each WAF deployment would require a fine-tuning to optimize the results. Furthermore, as presented in the section related to the criticalities encountered during the design and implementation phase (see section 8.6), ModSecurity has been slightly modified and, some CRS rules have been omitted. Lastly, the test has been run with the CRS paranoia level set to one, the minimum level. Another test that GoTestWAF provides is about false positives. It is implemented with a small number of requests (17), but an overall detection of 14 false-positive over 17 warns about the necessity of fine-tuning the WAF contextualizing it to the actual expected traffic and application that it has to secure. Further investigation about this high number of false-positive leads to payload containing sensitive keywords like bash or SQL commands. It may be expected that these elements trigger rules and, being Modsecurity deployed in the traditional mode detection, one triggered rule is enough to flag the request as malicious. The following is a list with some examples of false-positive payloads detected:

[6] https://github.com/wallarm/gotestwaf

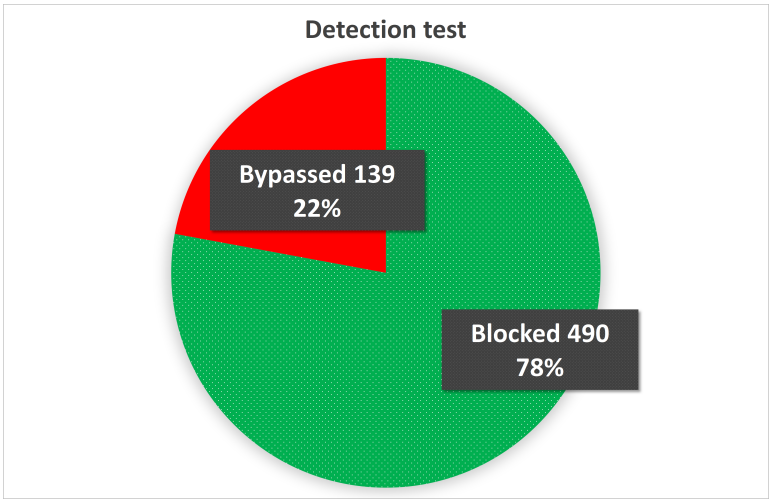

Figure 10.9: WAF WASM module security effectiveness test

```
union was a great select
ls 300 lexus
bash away in the gym
nc 8000 controller
```

Overall, these tests are nonetheless a satisfying indication of the functionality of a security control implemented as a WASM filter.

Chapter 11

Conclusions and future works

The objective of this work has been going towards a zero-trust network applied to Kubernetes, the de-facto standard for container orchestration in which more and more online services are migrating. This has been done by analyzing how the Istio service mesh can complement the already existing Kubernetes' security capabilities, adding effectively an additional layer of security.

Pursuing the defense in depth approach and exploiting the extension capabilities of Istio, a web application firewall based on ModSecurity and the OWASP Core Rule Set has been designed and developed. It allows testing the maturity of the extensibility technology implementing one of the most common and immediate security controls. The WAF permits to enforce L7 security features across the service mesh, filtering both the north-south traffic and the east-west one, that may come from compromised pods. Furthermore, Istio has been exploited to centralize the WAF management, configuring and deploying it across the mesh just via YAML configuration files. Envoy proxies that constitute the service mesh are unmodified and, at runtime, just receive the additional logic that has to be executed. The latter considerations are one of the main differences in respect to the related works described in section 8.1: with the proposed approach, Envoy does not have to be recompiled and distributed as a customized version, but rather, for any custom requirement, just the WASM can be edited, recompiled and easily distributed as an independent add-on. It also permits the application of the custom modules potentially to any Envoy deployment and to develop them in multiple languages, not sticking with Envoy's native one.

Tests in Chapter 10 shows that the technology stack used, based on WebAssembly, is actually mature enough to permit the WAF development, even if some limitations, mainly regarding access to the file system, are still in place. Other arisen limitations are in terms of stability of the WASM module, but they can be effectively compensated by leveraging the Kubernetes basic principles of horizontal scalability. Security features, especially CPU-intensive ones such as a WAF, still come with a cost. It is in terms of increased latency and resource usage. The proposed solution still provides high flexibility, potentially permitting to enable just the security rules needed for the specific workload, all via just configuration files. The alpha phase of the extensibility feature, and the limitation highlighted, show that WebAssembly modules are not yet production-ready, but it has been proved to be a working and promising technology stack to develop decentralized security controls in favour of zero trust architecture.

Future works can introduce further functionalities to the WASM WAF module, such as enabling the CRS Anomaly scoring mode and porting to WASM architecture other optional libModSec libraries. The latter, listed in Section 8.6, would lead to providing a more complete module with all the ModSecurity functionalities fully enabled. Furthermore, being WASM performance and proxy-WASM SDK areas of active development, they should be actively followed to overcome the current limitations above mentioned and go towards a stable, enriched and optimized WAF module.

Appendix A

User manual

The user manual is intended to provide a quick deployment guide of the proposed solution. Due to the fact that the whole design is based on the unmodified version of Kubernetes and Istio, the steps that have to be done to deploy the WASM WAF filter are minimal. for completeness, the steps to install both Kubernetes and Istio are provided. The whole manual has been tested on both Ubuntu 18.04.6 LTS and Ubuntu 20.04.3 LTS.

A.1 Kubernetes and Istio Installation

1. To install Kubernetes, K3s distribution is used: a lightweight distribution built for Internet of Things devices and edge computing[1]. Download K3s script and run it:

   ```
   curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="--no-deploy traefik" sh
       -
   ```

   Traefik proxy is used inside the k3s installation as the default ingress controller. However, due to the fact that on the top of Kubernetes we are going to deploy Istio, we will rely on it for ingress capabilities. Therefore, Traefik has to be disabled during the installation.

2. To verify that Kubernetes is up and running run:

   ```
   sudo systemctl status k3s
   sudo kubectl get nodes -o wide
   ```

   The expected output is one k3s node acting as `control-plane,master`.

   ```
   torsec-k3s Ready control-plane,master 47s v1.21.1+k3s1 192.168.0.84
       <none> Ubuntu 20.04.3 LTS 5.8.0-55-generic containerd://1.4.4-k3s2
   ```

3. Download Istio via the official script:

   ```
   curl -L https://istio.io/downloadIstio | sh -
   ```

4. Add *istioctl* to the system PATH

   ```
   export PATH=$PWD/bin:$PATH
   ```

5. Perform the installation precheck:

   ```
   istioctl experimental precheck
   ```

[1] https://k3s.io/

Troubleshooting: if istioctl is not capable of detecting Kubernetes installation and returns the error `dial tcp [::1]:8080: connect: connection refused` perform the following steps replacing `[name_of_your_host]` wth the name of your host.

```
cp /etc/rancher/k3s/k3s.yaml .
sed -i 's:localhost:hoseplak3s:;s:default:[name_of_your_host]:g' k3s.yaml
KUBECONFIG=~/.kube/config:k3s.yaml kubectl config view --raw > config.tmp
mv config.tmp /root/.kube/config
```

6. Finalize istio installation:

   ```
   istioctl manifest apply --set profile=demo
   ```

   Different profiles can be selected. `demo` is intended for tests providing the tracing of all the traffic and all the basic Istio elements (istiod,ingress and egress gateway) already installed. `default` profile does not install by default the egress gateway and traces only 1% of the traffic to reduce the performance overhead. Refer to the official documentation for the complete profiles comparison[2].

7. Now the injection label has to be added to the namespace that will be added to the service-mesh. In this example, the `default` namespace is used:

   ```
   kubectl label namespace default istio-injection=enabled
   kubectl get ns default --show-labels
   ```

8. Add-ons configuration files are provided inside Istio folder for an effortless installation of Kiali, Jaeger, Prometheus and Grafana.

   ```
   kubectl apply -f ./samples/addons/
   ```

9. Now that the services are deployed, is it possible to expose them via the ingress gateway. Run the following `expose-kiali.yaml` file via `kubectl apply -f expose-kiali.yaml` to expose kiali:

   ```
   apiVersion: networking.istio.io/v1alpha3
   kind: Gateway
   metadata:
     name: kiali-gateway
     namespace: istio-system
   spec:
     selector:
       istio: ingressgateway
     servers:
     - port:
         number: 80
         name: http-kiali
         protocol: HTTP
       hosts:
       - "kiali.k3s"
   ---
   apiVersion: networking.istio.io/v1alpha3
   kind: VirtualService
   metadata:
     name: kiali-vs
     namespace: istio-system
   spec:
     hosts:
   ```

[2] https://istio.io/latest/docs/setup/additional-setup/config-profiles/

```
  - "kiali.k3s"
  gateways:
  - kiali-gateway
  http:
  - route:
    - destination:
        host: kiali
        port:
          number: 20001
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: kiali
  namespace: istio-system
spec:
  host: kiali
  trafficPolicy:
    tls:
      mode: DISABLE
```

10. The service is now available connecting to `http://kiali.k3s`. For further access configurations, securing it and other configuration files to expose all the add-ons, refer to Istio documentation[3]. Note: it is needed to add the DNS entry inside the `hosts` file of the client pc. Under Ubuntu run the following command replacing `[NodeIP]` with the actual IP address of the node:

    ```
    sudo echo "[NodeIP] kiali.k3s" >> /etc/hosts
    ```

A.2 Use case scenario deployment

The scenario that is going to be deployed is the one already described in Use case chapter 9.

1. Apply the yaml configuration file already available inside the Istio folder:

    ```
    kubectl apply -f ./samples/bookinfo/platform/kube/bookinfo.yaml
    ```

2. Check that pods are up and running via `kubectl get pods`. Each of them has to have `2/2` containers. It means that both the main container and the sidecar container with istio-proxy (Envoy) are correctly running.

    ```
    NAME                             READY STATUS  RESTARTS AGE
    reviews-v2-7bf8c9648f-d4sqt      2/2   Running 0        104s
    reviews-v1-545db77b95-slsd5      2/2   Running 0        102s
    details-v1-79f774bdb9-cfjjt      2/2   Running 0        99s
    ratings-v1-b6994bb9-fbjr9        2/2   Running 0        98s
    reviews-v3-84779c7bbc-g8gcf      2/2   Running 0        98s
    productpage-v1-6b746f74dc-ghhk8 2/2   Running 0        22s
    ```

3. Deploy an ingress gateway to access the frontend service. Content of `gw.yaml`:

    ```
    apiVersion: networking.istio.io/v1alpha3
    kind: Gateway
    metadata:
    ```

[3] https://istio.io/latest/docs/tasks/observability/gateways/

```
  name: istions-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "istionuc.k3s"
---
```

Run:

```
Kubectl apply -f gw.yaml
```

4. Via a VirtualService link the requests coming from the ingress gateway to the frontend service, `productpage`. Content of `vsvc.yaml`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
  - "istionuc.k3s"
  gateways:
  - istions-gateway
  http:
  - match:
    - uri:
        exact: /productpage
    - uri:
        prefix: /static
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        prefix: /api/v1/products
    route:
    - destination:
        host: productpage
        port:
          number: 9080
```

Run:

```
Kubectl apply -f vsvc.yaml
```

5. Deploy a basic example of DestinatioRules. Content of `dst.yaml`:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: productpage
  namespace: istions
spec:
  host: productpage
```

```
  subsets:
  - name: v1
    labels:
      version: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
  namespace: istions
spec:
  host: reviews
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v3
    labels:
      version: v3
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ratings
  namespace: istions
spec:
  host: ratings
  subsets:
  - name: v1
    labels:
      version: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: details
  namespace: istions
spec:
  host: details
  subsets:
  - name: v1
    labels:
      version: v1
---
```

Run:

```
kubectl apply -f dst.yaml
```

6. check that everything has been correctly set. Opening `http://istionuc.k3s/productpage` the frontend page should be displayed.
   Note: just like in 9, it is needed to add the DNS entry inside the `hosts` file of the client

pc to access the exposed page via the ingress gateway. Under Ubuntu run the following command replacing `[NodeIP]` with the actual IP address of the node:

```
sudo echo "[NodeIP] istionuc.k3s" >> /etc/hosts
```

A.3 WASM WAF Filter deployment

1. Create `modsec-productpage.yaml` with the following content:

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: modsec-productpage
spec:
  workloadSelector:
    labels:
      app: productpage
  configPatches:
  - applyTo: HTTP_FILTER
    match:
      context: SIDECAR_INBOUND
      listener:
        filterChain:
          filter:
            name: envoy.filters.network.http_connection_manager
      proxy:
        proxyVersion: ^1\.*.*
    patch:
      operation: INSERT_BEFORE
      value:
        name: istio.modsec-productpage
        config_discovery:
          config_source:
            ads: {}
            initial_fetch_timeout: 0s # wait indefinitely to prevent bad
                Wasm fetch
          type_urls: [
              "type.googleapis.com/envoy.extensions.filters.http.wasm.v3.Wasm"]
---
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: modsec-productpage-config
spec:
  workloadSelector:
    labels:
      app: productpage
  configPatches:
  - applyTo: EXTENSION_CONFIG
    match:
      context: SIDECAR_INBOUND
    patch:
      operation: ADD
      value:
        name: istio.modsec-productpage
        typed_config:
          '@type': type.googleapis.com/udpa.type.v1.TypedStruct
```

```
type_url:
    type.googleapis.com/envoy.extensions.filters.http.wasm.v3.Wasm
value:
 config:
   vm_config:
     vm_id: vm_prodpg
     runtime: envoy.wasm.runtime.v8
     code:
       remote:
         http_uri:
            uri:
                https://github.com/M4tteoP/wasm-repo/raw/main/modsec.wasm
   # The configuration for the Wasm extension itself
   configuration:
     '@type': type.googleapis.com/google.protobuf.StringValue
     value: |
       {
         "modsec_config": [
          {
            "enable_default": "yes",
            "enable_crs": "yes",
            "enable_sqli": "yes",
            "enable_xss": "yes",
            "custom_rules":[
            "SecRuleRemoveById 920280"
            ]
          }
         ]
       }
```

2. Optionally customize the location of the deployment editing the label inside `workloadSelector` for both EnvoyFilters (default configuration will deploy it inside the istio-proxy of the productpage workload).
3. Optionally customize the Modsecurity rules provided to the WAF (default configuration enables the CRS). Refer to appendix section A.4 for details about Modsecurity configurations and to section 7.2.4 for basic information on how to write Modsecurity rules.
4. Apply the configuration file

   ```
   kubectl apply -f modsec-productpage.yaml
   ```

5. Check that the WAF WASM Filter is up and running performing two different requests:

   ```
   curl -I http://istionuc.k3s/productpage
   ```

   Is a legit request and `200 OK` is the expected return code, while:

   ```
   curl -I http://istionuc.k3s/productpage?arg=\<script\>
   ```

   Has to return `403 Forbidden` being a code injection attempt.

A.4 Modsecurity Configuration

One key element of this project is to provide enough flexiblity in terms of Modsecurity configuration without the necessity of recompiling each time the whole WASM file. This is achieved via the possibility of providing a JSON string inside the YAML file that is consumed by the WASM extension. The current JSON schema expected by the WASM filter is the following one:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "WASM Modsec configuration via YAML",
  "type": "object",
  "properties": {
    "modsec_config": {
      "type": "array",
      "items": [
        {
          "type": "object",
          "properties": {
            "enable_default": {
              "type": "string",
              "enum": [
                "yes",
                "no"
              ],
              "default": "yes"
            },
            "enable_crs": {
              "type": "string",
              "enum": [
                "yes",
                "no"
              ],
              "default": "yes"
            },
            "enable_sqli": {
              "type": "string",
              "enum": [
                "yes",
                "no"
              ],
              "default": "no"
            },
            "enable_xss": {
              "type": "string",
              "enum": [
                "yes",
                "no"
              ],
              "default": "no"
            },
            "custom_rules": {
              "type": "array",
              "items": [
                {
                  "type": "string"
                }
              ]
            }
          },
          "required": []
        }
      ]
    }
  }
```

```
}
```

A.5 Excluded CRS Rules

This section lists the ids of the rule excluded in the current WAF implementation. Alongside is also provided with the motivation behind the exclusion.

- `910100`: Detection of client IP from a high risk country, GeoIP library is needed.
- `913100`: Detection of User-Agent associated with security scanner, access to `scanners-user-agents.data` is required.
- `913110`: Detection of headers associated with security scanner, access to `scanners-headers.data` is required.
- `913120`: Detection of filename/argument associated with security scanner, access to `scanners-urls.data` is required.
- `913101`: Detection of User-Agent associated with scripts, access to `scripting-user-agents.data` is required.
- `913101`: Detection of User-Agent associated with crawlers, access to `crawlers-user-agents.data` is required.
- `930130`: Detection of access attempt to restricted files, access to `restricted-files.data` is required.
- `932120`: Detection of Windows Poweshell commands, access to `windows-powershell-commands.data` is required.
- `932160`: Detection of Unix shell commands, access to `unix-shell.data` is required.
- `932180`: Detection of restricted file upload attempt, access to `restricted-upload.data` is required.
- `933120`: Detection of PHP configuration directives, access to `php-config-directives.data` is required.
- `933130`: Detection of PHP variables , access to `php-variables.data` is required.
- `933150`: Detection of high-risk PHP functions, access to `php-function-names-933150.data` is required.
- `933151`: Detection of medium-risk PHP functions, access to `php-function-names-933151.data` is required.
- `944130`: Detection of suspicious Java class, access to `java-classes.data` is required.
- `951100`: Detection of SQL errors in response body, access to `sql-errors.data` is required.
- `952100`: Detection of Java source code leakages, access to `java-code-leakages.data` is required.
- `952110`: Detection of Java errors, access to `java-errors.data` is required.
- `930120`: Detection of OS file accesses attempt, access to `lfi-os-files.data` is required.

Appendix A

Developer manual

This appendix is intended to provide details about building from scratch all the components used inside this work and give technical details to maintain, customize and improve the WAF WASM project. The building phase is presented before diving deeper into the implementation details because it may have a broader interest. It can be used as guidance to build new WASM filters and provides examples on how to compile and link libraries compatible with the WASM architecture. On the other hand, the implementation details are more focused on the specific implementation describing programming decisions and how Libmodsecurity has been used.

A.1 Building phase

A.1.1 Building Libmodsecurity for WASM

WASM architecture is based on its own binaries format. It is therefore not possible to use libraries built for other architectures. Libmodsecurity and all its dependences have to be ported to the WASM architecture. The tool used is Emscripten, a compiler toolchain to WebAssembly that permits the porting of C/C++ projects to WASM[1].

1. Via `git` download and install Emscripten:

   ```
   git clone https://github.com/emscripten-core/emsdk.git
   cd emsdk
   git pull
   ./emsdk install 2.0.7
   ./emsdk activate 2.0.7
   source ./emsdk_env.sh
   ```

   Note: It is strongly suggested to maintain the same Emscripten version for both libraries and filter building. At the moment of writing, the WASM Filter building process relies on **Emscripten v 2.07**. Stick with this version or be aware that runtime errors may arise. The following section (A.1.2) provides details on how to check the Emscripten version used in the filter building toolchain proposed afterwards.

2. Download and install the WASI SDK, the modular system interface for WebAssembly[2].

   ```
   mkdir /opt/wasi-sdk
   cd /opt/wasi-sdk
   wget https://github.com/WebAssembly/wasi-sdk/releases/download/wasi-
   ```

[1] https://emscripten.org/

[2] https://docs.wasmtime.dev/wasm-c.html

```
sdk-12/wasi-sdk-12.0-linux.tar.gz
tar -xvf wasi-sdk-12.0-linux.tar.gz
export WASI_SDK_PATH="/opt/wasi-sdk-12.0"
```

3. Download the last dependecies via `apt`:

```
apt install libtool-bin automake texinfo
```

4. Download the PCRE library specifing the wasm-wasi branch and install it via the provided Shell script `build_for_crystal.sh`.

```
git clone https://github.com/maxfierke/libpcre.git -b
    mf-wasm32-wasi-cross-compile
cd libpcre
./build_for_crystal.sh .
```

5. PCRE library is now built as a `*.a` static library under `./targets/`.

6. Move it to `usr` folder to be afterwards reachable and inside the project folder under `/pcre/lib/`

```
mkdir /usr/local/pcre
cp ./targets/*.a /usr/local/pcre
cp ./targets/*.a ./basemodsec/pcre/lib
```

7. Download Libmodsecurity customized for WASM build and add `pcre.h` file.

```
git clone https://github.com/leyao-daily/ModSecurity.git
cd ModSecurity
cp /usr/include/pcre.h ./headers/
```

8. Built it with minimal dependencies:

```
./build.sh
git submodule init
git submodule update
emconfigure ./configure --without-yajl --without-geoip --without-libxml \
--without-curl --without-lua --disable-shared --disable-examples \
--disable-libtool-lock --disable-debug-logs --disable-mutex-on-pm\
 --without-lmdb --without-maxmind --without-ssdeep
     --with-pcre=./pcre-config
emmake make
emmake install
```

9. Libmodsecurity built for WebAssembly architecture is now under `/usr/local/modsecurity/lib`. Copy it and relative header filers inside the project.

```
cd ..
cp /usr/local/modsecurity/lib/* ./basemodsec/modsec/lib
```

The WASM module is now ready to be built relying on the libraries just compiled and added to the project's folder. The latter should now include:

```
basemodsec
├─ modsec
│  ├─ include
│  └─ lib
├─ pcre
│  ├─ include
│  └─ lib
└─ ...
```

A.1.2 Building the WASM Filter

The building process of a WASM Filter is based on Bazel, a open-source build tool that uses a high-level language to describe the build process of the project. The *WebAssembly for Proxies C++ SDK* has dependencies on specific tools versions, relying on Bazel, and on already available build rules files, permits to hide most of the complexity related to install the correct toolchain. The footnote link[3] shows the build function internally called. Rely on this file to check the version used in case you wish to perform manual builds. At the moment of writing **Emscripten v.2.0.7** and **Protobuf v.3.17.3** are used. If needed, the GitHub page of the SDK provides further building details[4].

1. Bazel can be downloaded via its wrapper Bazelisk:

   ```
   sudo wget -O /usr/local/bin/bazel \
   https://github.com/bazelbuild/bazelisk/releases/latest/download/bazelisk-linux-amd64
   sudo chmod +x /usr/local/bin/bazel
   ```

2. Via `apt` Install all the dependencies needed to build a C++ WASM extension with Bazel:

   ```
   sudo apt install gcc curl python3
   ```

3. Run the building process accessing the main folder of the project and running `bazel build`:

   ```
   cd ./basemodsec
   bazel build //:basemodsec.wasm
   ```

4. The wasm file is generated under `./bazel-bin/` folder.
   Note: Do not perform the bazel `build` command as root user otherwise the build will not end successfully.

Bazel build relies on two files that have to be properly configurated:

1. *WORKSPACE*: It is a textual file placed inside the source files directory. It is used to expand the project workspace adding external dependencies needed during the building process. It is done writing workspace rules. The most important rule used inside this project is `http_archive`:

   ```
   http_archive(
       name = "proxy_wasm_cpp_sdk",
       sha256 = PROXY_WASM_CPP_SDK_SHA256,
       strip_prefix = "proxy-wasm-cpp-sdk-" + PROXY_WASM_CPP_SDK_SHA,
       url = "https://github.com/proxy-wasm/proxy-wasm-cpp-sdk/archive/" +
           PROXY_WASM_CPP_SDK_SHA + ".tar.gz",
   )
   ```

   As the name states, `http_archive` rule is called to perform an http request that downloads an archive. Then, it is automatically decompressed and made available for binding. The previous snippet shows the `http_archive` rule called to download the complete SDK needed to develop in C++ proxy wasm filters. The other main statement used inside the WORKSPACE file of this project is `load`. It is used to import a symbol. The following snippet shows how `load` is used referring to the `proxy_wasm_cpp_sdk` archive just downloaded. Afterwards, the imported symbol is executed.

   ```
   load("@proxy_wasm_cpp_sdk//bazel/dep:deps.bzl", "wasm_dependencies")
   wasm_dependencies()
   ```

[3] https://github.com/proxy-wasm/proxy-wasm-cpp-sdk/blob/fd0be8405db25de0264bdb78fae3a82668c03782/bazel/dep/deps.bzl#L17

[4] https://github.com/proxy-wasm/proxy-wasm-cpp-sdk

Based on `http_archive` and `load` calls, the workspace of the project is populated with all the elements and dependencies needed to build a basic C++ WASM filter.

2. *BUILD*: BUILD file is a short program written in Starlark. It is a sequential list of statements used to write the build process and define the outputs. The statements are named *build rule functions* and are usually language specific functions. It is specified by the prefix (e.g. `cc_` means that it is specific for C++). The BUILD file of this project is made of three functions:

   (a) *Specification of the binary output*: Made via `wasm_cc_binary`. The name field is the name of the output file and it has to be used to call `bazel build` from the command line. Then the list of source files that can be both from the internal repository and external ones (e.g. `@io_istio_proxy`) and dependencies are listed in the arrays `srcs` and `deps` respectively.

   ```
   wasm_cc_binary(
       name = "basemodsec.wasm",
       srcs = [
           "plugin.cc",
           "plugin.h",
           "rules.cc",
           "rules.h",
           "@io_istio_proxy//extensions/common/wasm:base64.h",
       ],
       deps = [
           "@com_google_absl//absl/strings",
           "@com_google_absl//absl/time",
           "@io_istio_proxy//extensions/common/wasm:json_util",
           "@proxy_wasm_cpp_sdk//:proxy_wasm_intrinsics",
           ":libpcre",
           ":libmodsecurity",
       ],
       copts = [
       "-Imodsec/include/",
       "-Ipcre/include/"
       ],
   )
   ```

   (b) *Inclusion of the precompiled Libmodsecurity library*: `cc_library` build rule function permits to import precompiled C/C++ libraries. Therefore it can be used to import the libraries compiled in appendix A.1.1. Once defined a name, we must provide the path to the static library and the full list of header files. Finally, setting `visibility` to public permits the library to be used by targets in other packages.

   ```
   cc_import(
       name = "libmodsecurity",
       static_library = "modsec/lib/libmodsecurity.a",
       hdrs = [
           "modsec/include/modsecurity/actions/action.h",
           "modsec/include/modsecurity/collection/collection.h",
           "modsec/include/modsecurity/collection/collections.h",
           [...]
           "modsec/include/modsecurity/transaction.h",
       ],
       visibility = ["//visibility:public"],
   )
   ```

   (c) *Inclusion of the precompiled libpcre library*: just like Libmodsecurity, via `cc_library` libpcre library is also imported.

```
cc_import(
    name = "libpcre",
    static_library = "pcre/lib/libpcre.a",
    hdrs = [
        "pcre/include/pcre.h",
    ],
    visibility = ["//visibility:public"],
)
```

A.2 Implementation details

The source code files are:

- `plugin.cc` and `plugin.h` with the whole logic of the application.
- `rules.cc` and `rules.h` with the basic configuration rules and CRS rules hardcoded.

About the application logic, to implement a WASM module, the following elements are mandatory:

- *Implementation of the root context class*: Named `PluginRootContext`, which inherits the `RootContext` class defined inside the SDK. The root context object is created during the bootstrap of the WASM module and has the lifetime of the VM on which the module is executed. Here reside all the elements that have to be kept alive across requests. This is the case of three core elements:
  - `modsecurity::ModSecurity *modsec`: `modsec` points to a `ModSecurity` object, allocated during the initial configuration of the WASM module. It is the main ModSecurity element that, within the following `rules` object, will be used to initialize a transaction and perform security controls every needed time.
  - `modsecurity::RulesSet *rules`: It is another object from libmodsecurity. It points to a `RulesSet` object that gets populated with the rules that will be used by the transaction.
  - `PluginRootContext::ModSecConfigStruct modSecConfig`: It is a custom `struct` used to keep all configuration elements received from the configuration file. It is made of boolean variables about enabling or not the default configurations, the CRS and specific parts of it (for SQLi and XSS detection). It also includes a vector of strings for the custom rules. Its declaration is:

    ```
    struct ModSecConfigStruct {
      bool enable_default;
      bool enable_crs;
      bool detect_sqli;
      bool detect_xss;
      std::vector<std::string> custom_rules;
    };
    ```

- *Implementation of the stream context class*: Named `PluginContext`, which inherits the `Context` class defined inside the SDK. A context object is created for each steam and is deallocated once the stream itself ends. It is therefore possible to rely on this object just between events of the same stream. The key element that requires to have a visibility of the whole stream is the `Transaction` object of libmodsecurity. It is named `modsecTransaction` and is allocated when a stream begins, as soon as the request headers are handled by the module. As the following snippet of code shows, its instantiation relies on the two elements previously explained in the root context:

  ```
  modsecTransaction = new modsecurity::Transaction(rootContext()->modsec,
      rootContext()->rules, NULL);
  ```

- *Override context API methods to handle events*: Inside the `PluginContext`, API methods are also defined. They correspond to the callbacks for stream events already introduced during the design analysis (see Section 8.5.2). Describing them in a more technical way, the full list of overridden functions for this project are:
  - `FilterHeadersStatus onRequestHeaders(uint32_t, bool)override`: Here the transaction object is created and, via the function `initTransaction`, is populated with basic information of the connection (client IP, client port, destination IP, destination port) and of the request (url and method). Once `modsecTransaction` is populated `process_intervention` is applied to analyze the content. Because most of the CRS rules are not applied at phase 1 (see Modsecurity chapter 7), a little trick is coded: an empty body is added to the transaction. Doing so, the transaction enters phase 2 and the rules are correctly applied.
  - `FilterDataStatus onRequestBody(unsigned long, bool)override`: `onRequestBody` reads the body of the request and adds it to `modsecTransaction`. Afterwards, the process function on the request body is applied (function `processRequestBody`).
  - `FilterHeadersStatus onResponseHeaders(uint32_t, bool)override`: same as the previous `on RequestHeaders`, but applied on the headers of the response.
  - `FilterDataStatus onResponseBody(unsigned long, bool)override`: same as the previous `onRequestBody`, but applied on the headers of the response.
  - `void onDelete()override`: the `onDelete` calback is triggered when the stream is ended and the stream context is up for deconstruction. Here the `Transaction` object pointed by `modsecTransaction` is deallocated. Skipping this action leads to leak the memory pointed and to a fatal state of the WASM filter. It is described in more details in the test chapter, in Section 10.2.

  The `PluginRootContext` includes the API methods for initialization events. Here, the overridden function is just one:
  - `bool onConfigure(size_t)override`: executed just at the startup, after the creation of the root context. It instantiate the `modsec` and `rules` object, parse the JSON object received from the configuration file and populate accordingly both the `modSecConfig` structure and the `rules` object.
- *Register the root context and stream context*: It is done at the beginning of the `plugin.cc` file with the following static function:

```
static RegisterContextFactory
    register_Example(CONTEXT_FACTORY(PluginContext),
    ROOT_FACTORY(PluginRootContext))
```

Two other functions that worth to be mentioned are `alertActionHeader` and `alertActionBody`. Each `process_intervention` executed inside the callbacks returns a number that, if it is different from `0`, means that a rule has been matched. Implementing the rules in the traditional mode, one rule is enought to stop the chain and return an error code to the client. It is accomplished by these functions, that, with minimal differences between them, are as the following one:

```
FilterHeadersStatus PluginContext::alertActionHeader(int response){
    sendLocalResponse(403, absl::StrCat("Dropped by alertActionHeader
        response= ",std::to_string(response)), "", {});
    return FilterHeadersStatus::StopIteration;
}
```

The `sendLocalResponse` function, defined inside the C++ SDK, permits to send a response back to the client. `403` explicit the return code. Small differences happen based on when the `alertAction` is triggered. Stopping the stream at phases 1, 2 or 3 leads to the expected behaviour, with a 403 error page. But, due to the no buffering phases described in section 8.5.2, if a rule is matched at phase 4, the response header is already sent to the client. Therefore, only the body response will be blocked. The client will receive a status `200` with an empty body.

A.3 Debugging Tips

This section provides tips while debugging a WASM filter.

- *Change Envoy log level*: by default, Istio injects the istio-proxy (Envoy) with log levels set as `info`. `trace` and `debug` are more verbose alternatives, and can be set:
  - Performing the manual injection of the sidecar with log level properly configured inside inject-values.yaml[5].
  - Via `istioctl proxy-config` on a specific proxy already deployed: `istioctl pc log pod_name.<namespace> --level wasm:trace`. Replace `pod_name` and `<namespace>` (omit if it is `Default`) respectively with the name of the pod and the namespace in which it is deployed.
- *`kubectl` logs*: Reading the logs provided by kubectl from the istio-proxy container is the main source of logs. To analyze burst of traffic it is possible to redirect the output directly to a file: `kubectl logs -f pod_name -c istio-proxy -n namespace_name > logs.txt`.
- *`dmesg` from istio-proxy pod*: executed from inside the istio-proxy container, `dmesg` command may provide some hints about crashes.
- *Monitor resources* via:
  - `crictl stats` directly providing the id of the sidecar container (retrive it via `crictl ps`).
  - Grafana Dashboard exposing the service with Istio[6] and analyzing the pre-made *WASM Extension Dashboard*.

[5] https://istio.io/latest/docs/setup/additional-setup/sidecar-injection/#manual-sidecar-injection

[6] https://istio.io/latest/docs/tasks/observability/metrics/using-istio-dashboard/

Bibliography

[1] A. Van Cleeff and R. J. Wieringa, "Rethinking de-perimeterisation: Problem analysis and solutions", IADIS International Conference Information Systems 2009, Barcelona (ES), 2009, pp. 105–112

[2] S. W. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero trust architecture", August 2020, DOI 10.6028/NIST.SP.800-207

[3] S. Vikas, K. Gurudatt, M. Vishnu, and K. Prashant, "Private vs public cloud", International Journal of Computer Science & Communication Networks, vol. 3, no. 2, 2013, p. 79. https://www.researchgate.net/publication/258253155_Private_Vs_Public_Cloud

[4] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture", International Journal of Open Information Technologies, vol. 2, no. 9, 2014, pp. 24–27. https://www.academia.edu/download/36701854/micro.pdf

[5] H. Tabrizchi and M. K. Rafsanjani, "A survey on security challenges in cloud computing: issues, threats, and solutions", The journal of supercomputing, vol. 76, no. 12, 2020, pp. 9493–9532. https://link.springer.com/article/10.1007/s11227-020-03213-1

[6] J. Tang, Y. Cui, Q. Li, K. Ren, J. Liu, and R. Buyya, "Ensuring security and privacy preservation for cloud data services", ACM Computing Surveys (CSUR), vol. 49, no. 1, 2016, pp. 1–39, DOI 10.1145/2906153

[7] P. Rana and I. Batra, "Detection of attacks in cloud computing environment–a comprehensive review", 2021 2nd International Conference on Intelligent Engineering and Management (ICIEM), London (UK), 2021, pp. 496–499, DOI 10.1109/ICIEM51511.2021.9445284

[8] Kubernetes project, https://kubernetes.io

[9] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices", 2020 IEEE Secure Development (SecDev), Atlanta (GA, USA), 2020, pp. 58–64, DOI 10.1109/SecDev45635.2020.00025

[10] S. Cha and H. Kim, "Detecting encrypted traffic: a machine learning approach", International Workshop on Information Security Applications, Jeju Island (Korea), 2016, pp. 54–65, DOI 10.1007/978-3-319-56549-1_5

[11] D. D'Silva and D. D. Ambawade, "Building a zero trust architecture using kubernetes", 2021 6th International Conference for Convergence in Technology (I2CT), Pune (India), April 2021, pp. 1–8, DOI 10.1109/I2CT51068.2021.9418203

[12] Istio project, https://istio.io

[13] Envoy proxy project, https://www.envoyproxy.io

[14] WebAssembly project, https://webassembly.org

[15] ModSecurity project, https://www.modsecurity.org

[16] J. G. Lara and À. P. Gracia, "Building web application firewalls in high availability environments", Web Application Security, vol. 72, pp. 75–82, Springer, 2010, DOI 10.1007/978-3-642-16120-9_18

[17] OWASP ModSecurity Core Rule Set project, https://coreruleset.org