POLITECNICO DI TORINO

Master’s Degree in Computer Engineering

Master’s Degree Thesis

Zero Trust Network Security Model in Containerized Environments

Supervisor:
Prof. Cataldo Basile

Candidate:
Alessio Dongiovanni

Company Supervisor:
Andrea Buonerba
Salvatore Pecoraro
Stefano Loscalzo

Academic Year 2023/2024
Torino

Acknowledgements

Thanks to Spike Reply for giving me the opportunity to do this thesis work. Thanks to Salvatore, Stefano and Giorgio, who supported me throughout the analysis and implementation process. Thanks to Prof. Basile for his supervision, and for the passion he transmitted in this discipline.

A huge thanks to my mother and my father, for all the love and support they have given me over the years and for every single sacrifice they have made to allow me to reach this great milestone today. I couldn't have asked for better parents in life. I owe you everything. I love you.

Thanks to my sister, to whom I have never dispensed love, but for whom I would actually give all of myself. I will always be at your side. I love you (although I'll never explicitly admit it).

Thanks to my grandparents, for all the love I received from them and for everything they passed on to me, which I always carry with me. Your memory will always remain among the most beautiful ones, to be jealously guarded. You will always be a part of me, wherever I am.

Thanks especially to my grandma Tetta, my second mum, whom I am lucky enough to still be able to hug and see smile. Lunches and dinners with you are always the best thing, and they keep alive part of that child you saw growing up. You don't know how poignant it is to say goodbye to you and see you sad every time I leave, hoping to be able to hug you again soon. Thank you for everything, I love you immensely.

Thanks to Martina, who has always stood by me and believed in me even when I did not and wanted to give up. Thank you also for all the peacefulness and lightness that you always manage to transmit to me. I could not have asked for a better person at my side. Here's to many more successes to celebrate together. I love you.

Thanks to Ari and Niko, sister and brother not by blood, who taught me what true friendship is, and who despite the distance and the years continue to be by my side, always. May it last forever, no matter what comes.

Last but not least, a big thank you to all my lifelong friends who have stood by me over these years. To Gabri P. with whom we grew up side by side, to Gabri C., Enrico, Stefano, Giulio and all the others. Thanks to our friends in Turin, who made this long journey less burdensome. Thanks to Giulia, who has only been in my life for a short time but whom I love as if I had known her for years. Finally, thank you to those who have been in my life even in the slightest over these years, leaving me something that, both positive and negative, have made me the person I am.

Table of Contents

Chapter 1

Introduction

Over the past decade, the evolution of the technological landscape has been characterized by a constant search for increasingly efficient, scalable and flexible IT solutions, moving from centralized and monolithic systems to distributed and cloud-oriented models. The impact of cloud computing services and today's business needs to be agile, responsive and able to manage and scale increasingly complex workloads have led to the development and adoption of new cloud-native patterns, technologies and tools. As a result, microservice architecture and containers, along with orchestration platforms such as Kubernetes, have emerged as leading transformation technologies that enable organizations to meet these needs. Modern cloud-native applications are designed and built on a microservice architecture composed of hundreds of small services that communicate and cooperate to form a meaningful application. The old, traditional monolithic approach to application development is thus being abandoned in exchange for greater efficiency in developing, managing, maintaining, and scaling individual services independently of others.

Nevertheless, virtualization still remains a key technology enabling cloud computing and data centers, but the use of traditional virtual machines to deploy and run applications is currently being replaced by the adoption of a new virtualization technology called containers. Containers are lightweight, properly isolated software packages, each of which encapsulates the application, its dependencies and necessary libraries, providing consistent portability between development and production environments, lower resource consumption and increased scalability. Containers are a good way to bundle and run your applications. However, adopting containers for microservices architecture involves running a large number of containerized entities, and as a result, manually managing and scaling these entities, ensuring no downtime, has become quite complex. This is why the advent of Kubernetes has been so successful. Also regarded as "the operating system of the cloud," Kubernetes has become the leading orchestrator adopted for deploying and managing cloud-native applications. Completely open source, K8s can run on any type of cloud, on-premise data center or hybrid models, abstracting the differences between the underlying infrastructures and thus enabling easy migration between different cloud platforms. It simplifies the operational complexity of modern distributed systems and improves their flexibility and resilience at runtime by taking care of the scalability and failover of your application in a fully automated way, and providing many other useful features. According to Gartner's prediction, by 2027 more than 90% of global organizations will be running containerized applications in production, and 64% of them are already adopting K8s as container orchestrators in production, while 25% is evaluating or piloting its adoption in the future. [1] [2]

But today's continuous increase in cyber attacks outlines a rapidly evolving threat landscape.

Considering that cyber attacks can lead to devastating losses of money, trust and reputation, companies are inherently motivated to strengthen their security infrastructure. Moreover, regulatory requirements are increasingly calling for tailored protections for different data categories, demanding both technical and organizational measures to ensure data confidentiality, integrity, availability, and privacy. However, as digital transformation advances, the cybersecurity scenario is becoming more complex and challenging. Indeed, companies' infrastructures where services and data reside are becoming increasingly complex, often consisting of a combination of several internal networks with a large number of endpoints, remote offices and workstations, mobile devices and cloud services. Relying upon IaaS, PaaS, and SaaS services, organizations are migrating their infrastructure, platforms, and applications (or even part of them) to cloud. As a result, modern enterprise digital infrastructures are no longer confined within a single perimeter that clearly separates them from the rest, but span on-premises data centers as well as private, public and hybrid clouds. The Covid pandemic added further complexity to the cybersecurity scenario by introducing remote working, which remains widely used by companies to this day.

This has rendered traditional perimeter-based network security solutions, such as firewalls, intrusion detection systems, and others, inadequate because there is no longer a single, easily identifiable perimeter for the enterprise to protect. Moreover, because these types of security measures focus on north-south traffic, once attackers breach the perimeter there is no visibility into east-west traffic and therefore unauthorized lateral movement cannot be prevented. Finally, containerized environments have introduced additional complexity into the scenario and challenges for enterprise IT security, as containers managed by Kubernetes are ephemeral entities that continuously appear and disappear, thus dynamically changing the IP address used, which complicates network security management and implementation of access control policies. The inadequacy of the 'castle and moat' approach and the difficulties in defining the perimeter of an organisation's information systems had already been highlighted in 2003 by the Jericho Forum and later taken up by John Kindervag, a researcher at Forrester, who in 2010 proposed as a solution a more rigorous approach to cybersecurity and access control within companies, called Zero Trust. The Zero Trust security model assumes that all entities within the network could potentially be compromised and therefore their operations must never be implicitly trusted, and every communication must be protected regardless of network location. Implementing Zero Trust principles also means that each resource must continuously undergo a security posture assessment through a Policy Enforcement Point before a request for access to a company-owned asset is granted, adopting the principle of least privilege access, and that full visibility over the network requests and traffic is achieved.

The goal of this thesis, developed through the support of Spike Reply company, was to demonstrate how Zero Trust can be achieved in Kubernetes environments, facing the complexity introduced by the Kubernetes network model, which makes traditional security solutions being unable to inspect and correctly understand the actual traffic exchanged, neither the east-west nor the north-south one: in fact, containers communications in K8s occur trough overlay networks that make use of IP-IP or VXLAN tunnels, as well the virtual network internal to the nodes and the host source/destination NAT. Therefore, it is clear how Kubernetes network security differs from traditional IT and infrastructure systems: many new aspects open the door to possible new threats and attacks, easily leaving room for bad actors to infiltrate and move laterally through the network and sensitive workloads if appropriate security measures are not taken.

A Proof of Concept was then accomplished by implementing a microservices application through SpringBoot and then deploying it on a Kubernetes cluster made of 3 nodes: one master node and two worker nodes. The application realizes a trivial Transport Ticketing System which consists of 4 microservice: the Login service, the Traveler service,the Catalogue Service, and the

Payment Service. Each microservice comes with its own private Postgres database instance for storing and retrieving data, and cooperates with at least one microservice to implement some specific functionalities.

Afterwards, two possible solutions for implementing zero-trust network principles within Kubernetes environments have been evaluated: the Istio Service-Mesh and the Palo Alto CN-series container-based Next-Generation Firewall. Therefore, in Chapters 6 and 7, the capabilities of both solutions were analyzed and tested, in order to understand which features are actually useful to achieve Zero Trust and obtain full visibility and control over network activities within the cluster. Finally, Chapter 8 outlines the final considerations you should take into account when using these two tools to implement Zero Trust security within your network.

Chapter 2

Background Concepts

In this chapter, we are going to explore and understand what approaches, paradigms, and leading technologies are driving digital transformation.

We will see why microservice architectures are becoming increasingly popular, and what great benefits the use of containers and orchestrators such as Kubernetes introduce. Next, we will go over a brief overview of the basic concepts of the Kubernetes architecture and its components, focusing on the related resources employed in this thesis work.

2.1 Microservice Architecture

Before the cloud revolution and microservice era, the traditional way of designing and developing 'standard' web applications was purely monolithic. With this architectural approach, any web application is conceived as a monolith, that is, a set of logical layers and functionality that are all implemented and executed in a single process. By adopting this approach, a web app is, in general easy to design and implement, but it presents significant challenges in terms of scalability, complexity, fault tolerance, and agility, especially as applications grow in size and complexity. But this does not meet today's business needs, which continually require significant innovations in web systems to be delivered quickly, frequently and reliably, as well as a high level if scalability, flexibility and resilience. And this is the reason why microservice architectures have become so popular nowadays.

Microservices - also known as microservice architecture - is an architectural paradigm that relies on breaking down a monolithic application and its functionality into a series of loosely coupled and independently deployable services. These services have their own business logic and database (keeping separated and private their own data) with a specific goal, following the *divide et impera* principle: complex systems can be spitted in smaller parts until they are small enough to be easy understandable. Microservices do not reduce complexity, rather they decompose it and make it easier to manage by separating tasks into smaller processes that operate independently of each other and contribute to the overall capabilities of the system.

Of course, in order to provide the functionalities of a web application, these microservices require some form of communication and cooperation; each one is in charge of managing a given business domain area, implementing part of the web app functionalities. Indeed, often a service executes a part of a larger, more complex request that has been previously broken down generally by a front-end layer. Often a service acts as a client of other microservices, contacting their APIs

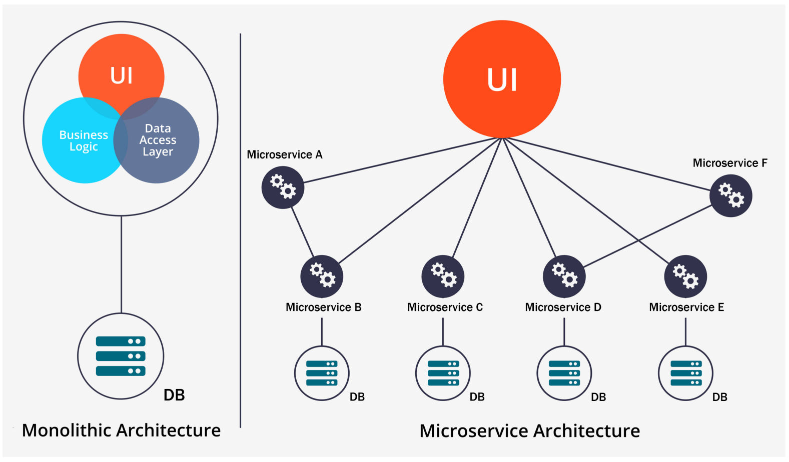


Figure 2.1: Monolithic vs Microservices Architecture (source: Hengky Sanjaya Blog)

and requesting certain functionality from them to accomplish a certain task and complete a more complex request it has received. [3][4]

Figure 2.1 shows the difference between a web application developed by adopting a monolithic approach and a microservice-oriented one. it is clear how a monolithic app encapsulates any service and functionality in a single process, adopting a single DB instance (perhaps replicated multiple times). Conversely, microservices architecture involves breaking down the application business logic that resides behind the front-end tier into a multitude of smaller services, each with its own application logic and DB instance (i.e. private data) separate from the others. Some microservices interact with others to elaborate and provide a more complex task, and some do not. Communication can be synchronous or implemented asynchronously by means of a message broker such as Apache Kafka, which I used to implement the test microservice architecture in Chapter 5.

Microservice architecture offers numerous benefits and advantages that meet modern software development and deployment requirements:

- *Flexibility*: Each microservice can be developed, deployed, and maintained independently, allowing teams to use different technologies, frameworks, and databases according to their needs.

- *Scalability*: Microservices enable individual services to scale independently when they reach their load capacity. Compared with scaling the entire application when the amount of requests increases, this granular scalability ensures optimal resource utilization and cost-effectiveness.

- *Resilience*: By distributing functionality across multiple services, microservice architectures enhance the resilience of the system. A single failure of a given service does not impact the entire system, but is contained within the individual service and can be easily mitigated through microservice redundancy, failover mechanisms, and graceful degradation strategies.

- *Agility*: Microservices support faster development cycles and more frequent deployments, facilitating a continuous integration and delivery strategy. They also simplify debugging, troubleshooting, and updating software, reducing testing and maintenance costs. This agility enables organizations to respond quickly to changing market demands and customer feedback.

But microservice architecture also comes with some challenges to face:

- *Design and Development Complexity*: Decomposing an application into smaller, more specialized services requires a high level of expertise and careful planning and management of dependencies, communication protocols, and data consistency within the same transaction.

- *Service Coordination*: Microservices need to communicate and coordinate effectively, requiring mechanisms for service discovery, load balancing, fault tolerance, and inter-service communication. Coordinating a large number of services can become challenging, especially as the system evolves and scales.

- *Operational Overhead*: Operating and managing a microservices-based system involves managing a large number of services, each with its own deployment, monitoring and scalability requirements. This can result in increased operational costs and complexity, including managing infrastructure resources, configuring deployment pipelines, and diagnosing and troubleshooting problems in a distributed environment. Implementing robust DevOps practices and automation tools is essential to effectively manage the operational challenges of microservice architecture.

- *Security*: The microservice architecture also presents some security challenges related to the large number of distinct entities operating. The first is the increased overall attack surface, since each microservice represents a potential entry point for attackers. Managing authentication and authorization across services and ensuring consistent application of security policies can be challenging. Ensuring the confidentiality and integrity of sensitive data distributed across microservices becomes crucial. Encryption and access control techniques should be implemented to protect data both in transit and at rest. Securing communication between microservices is essential to prevent eavesdropping, tampering, and unauthorized access. Finally, monitoring and logging microservices activities are essential to detect and respond to security incidents, but they become challenging in a distributed system composed of many entities.

Microservices continue to gain popularity due to the the innate flexibility of this architecture in the cloud. As the size and scope of cloud-based applications and workloads continue to grow, it is increasingly difficult and time-consuming to adapt monolithic architectures to meet new business needs. Microservices and cloud-native patterns share common principles and practices that enable organizations to build scalable, resilient, and agile applications in modern cloud environments, while leveraging the scalability, flexibility and automation provided by cloud platforms. In fact, a recent study conducted by Gartner revealed that nearly three-quarters (74%) of the organizations surveyed are currently using microservice architecture. 23% of the leaders' organizations are not yet doing so but plan to do it soon. [5].

In the next sections of this chapter, we will dive into cloud-native virtualization and orchestration technologies, which are widely used today and are well suited to microservices architecture, allowing their full potential to be exploited. Instead, the microservices security challenges mentioned above will be reconsidered and addressed in Chapters 4 and 6.

2.2 Virtualization and Containers

In recent years, digital transformation and the bursting advent of cloud computing have revolutionized the way applications are deployed and managed. One of the key innovations driving this transformation is the adoption of containerization technology.

Until a few decades ago, organizations relied on physical servers to host and run their applications. But at some point, this approach showed some limitations for organizations and data centers, as it did not allow clear boundaries to be established for each application's resources, leading to resource allocation and security issues.

When multiple applications share a single physical server, one application often monopolizes resources, causing performance problems for the others. Or even worse, a security incident affecting one application could threaten and impact the others as well.

For this reason, companies began to run each application on different physical servers, but this solution was not scalable because resources were underutilized and it was expensive for companies to maintain many physical servers. [6][7]

The winning solution has been the introduction of virtualization. Widely adopted by data centers and enterprises, it has so far been the ideal solution for running applications in isolated environments. Virtualization technology allows multiple virtual machines (VMs) to run on the same physical server completely isolated from the others, each with its own separate operating system, dependencies and resources.

It enables the isolation of applications running on separate virtual machines within the same physical server, and provides a great level of security, as the resources, processes/threads and information of one application cannot be freely accessed by another running on the same physical host. The strong isolation property of VMs provides a great level of security, as the resources, processes/threads and information of one application cannot be freely accessed by another VM running on the same physical host. Virtualization offers many benefits, such as increased scalability, better utilization of physical resources, and reduced resource consumption and costs.

Virtual machines have been around for decades and enable companies to deploy several servers running different applications on a single physical machine, even if they are running different operating systems.

Despite the advantages introduced by virtual machines, they still have some limitations that do not meet current business needs. Indeed, each virtual machine requires the installation and boot of its own full operating system (OS), resulting in resource overhead, and limited portability because moving virtual machines to different environments, such as from on-premise to the cloud or between different cloud providers, can be complex and time-consuming due to differences in infrastructure and basic configurations. In addition, virtual machines take several minutes to spin up and down, limiting their scalability capabilities. [6][7]

On the other hand, containers have rapidly gained popularity in the early 2010s with the development and diffusion of technologies such as Docker, and their features have quickly become attractive for solving the limitations of VMs.

Containers are still a virtualization technology, but they take a different approach to application deployment. Instead of virtualizing the entire operating system on which software runs, containers encapsulate only applications in lightweight, portable units. This allows applications to run consistently in different environments without the overhead associated with virtual machines.

This technology is based on the concept of containerization, which involves packaging an

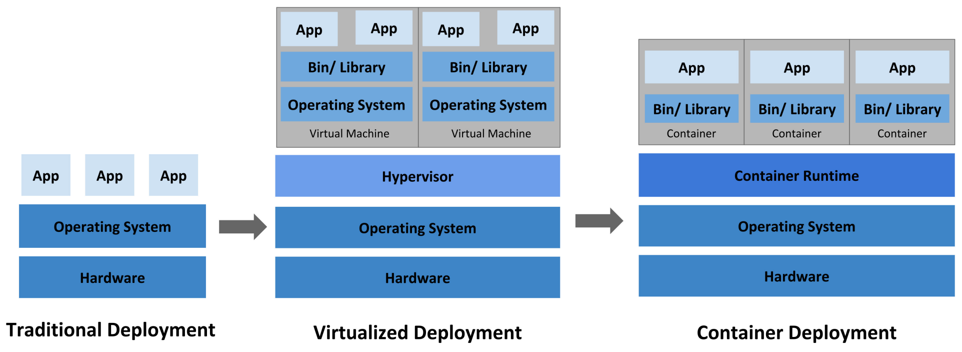


Figure 2.2: Evolution of Application Deployment (source: Kubernetes Documentation)

application along with its dependencies into a single unit called a container. Each container runs as an isolated process, sharing the host operating system with other containers. This lightweight approach to virtualization allows containers to boot quickly and consume fewer resources compared to VMs, since it is required to boot only a single OS for running many containerized applications.

With containers, developers can package their applications once and run them anywhere, from development environments to production servers and cloud environments. [6][7]

2.2.1 Linux Namespaces and Container Networking

The Linux kernel comes with various features intentionally provided to provide multi-tenancy on hosts. Linux namespaces, among others, are the fundamental technologies behind modern container implementations. They provide the highest level of isolation of global system resources between independent processes running on the same operating system, which is useful for achieving a more efficient and fine-grained level of virtualization. Given two processes running within separate containers on the same server, without namespaces, they would be able to interfere with each other's resources, leading to the resource management and security problems mentioned when discussing the pre-VMs era. There are several namespaces used to build containers, one for each different type of resource they aim to isolate [8]. By namespacing these resources, the process in a container isn't even aware that the processes in other containers exist. Here is a list of the several Linux namespaces types and what they aim to isolate:

- *Mount Namespace*: isolate filesystem mount points
- *UTS Namespace*: isolate hostname and domain name
- *IPC Namespace*: isolate interprocess communication (IPC) resources
- *PID Namespace*: isolate the PID number space
- *Network Namespace*: isolate network interfaces
- *User Namespace*: isolate UID/GID number spaces
- *Cgroup Namespace*: isolate cgroup root directory

We can think of containers as applications whose resources are isolated by a series of layers that make up the various namespaces. Therefore, each container typically has its own network namespace, providing isolation and allowing it to have its own virtual interfaces, including network stack, IP addresses and routing tables. Containers can be connected to one or more virtual or physical networks using various networking technologies such as bridges, overlays, or directly attached networks. These networks facilitate communication by allowing containers to send and receive data packets using standard networking protocols like TCP/IP.

Container orchestrators like Kubernetes facilitate the networking configuration and ensure connectivity between containers, handling tasks such as network interface creation and its IP address allocation, service discovery, and load balancing to support the seamless operation of containerized applications.

2.2.2 Advantages of Containers

Containers offer several advantages over traditional virtual machines, especially in the context of cloud computing and digital transformation:

- *Efficiency*: Containers are lightweight and consume fewer resources compared to VMs, making them ideal for deploying and scaling applications in cloud environments. Indeed, containers spin up in milliseconds since they do not require to boot of any OS. Moreover, a single system can host many more containers as compared to VMs.
- *Cloud and OS Distribution Portability*: Containers encapsulate applications and their dependencies together, enabling consistent deployment across different environments and OS, from on-premises data centers to public and hybrid clouds. Each container runs the same on a laptop as it does in the cloud. It also makes it easy to migrate workloads from any platform to one another.
- *Scalability*: Containers can be easily scaled up or down to handle dynamic workloads, enabling organizations to optimize resource utilization and improve application performance.
- *Isolation*: Containers offer process-level isolation, ensuring that applications run in isolated environments without interfering with each other. This enhances security and helps prevent conflicts between applications.
- *Continuous development, integration, and deployment*: Containers provide for reliable and frequent containerized application image build and deployment with quick and efficient rollbacks.

Thus containers have emerged as a transformative technology to modernize application deployment and management in cloud environments. Since containers can run on top of a shared OS, they need only to include application code, whether in the form of a single monolithic application or microservices that are bundled together in one or more containers to encompass a business function. By leveraging containerization technology, organizations can achieve greater efficiency, flexibility, and scalability, driving innovation and accelerating digital transformation initiatives.

It is clear how the benefits introduced by containers match well with the requirements and advantages of microservices, providing technological support that facilitates the implementation of such an architecture: rather than deploying many microservices in separate virtual machines, it is better to do so in separate containers. As the adoption of microservices continues to grow, containers are becoming an essential building block for building and deploying cloud-native applications in today's dynamic and competitive landscape. [7]

According to Gartner's prediction, by 2027, more than 90% of global organizations will be running containerized applications in production, which is a significant increase from fewer than 40% in 2021. [1]

However, considering the large number of microservices and containers that an enterprise must deploy and run, manually managing such a large number of entities (starting, restarting, terminating, scaling, etc.) becomes complex and challenging. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. Therefore, container orchestration platforms, such as Kubernetes, clearly become even more essential by automating the deployment, scaling, and management of containers, streamlining the application lifecycle and reducing operational overhead. Kubernetes provides a framework for running distributed systems in a resilient manner. It addresses application scalability and failover, provides deployment models and more. [6]

We will look at Kubernetes and its features in more detail in the next sections.

2.3 Kubernetes

In recent years, Kubernetes has emerged as the de facto open-source standard for container orchestration, revolutionizing the way organizations deploy, manage and scale containerized applications. Before Kubernetes, managing containerized applications at scale was complex and challenging. Container orchestration involved manually scheduling and managing containers across a cluster of machines, handling resource allocation, scaling, load balancing, and fault tolerance. As the adoption of containers - and thus the number of containers to be managed - increase, this manual approach was time-consuming, error-prone, and difficult to scale, especially in dynamic cloud environments.

Kubernetes, also known as K8s, addresses these challenges by providing a robust and comprehensive platform for automating containerized application deployment, scaling, monitoring, and management, as well as rolling out changes to your apps, and other interesting features. Moreover, Kubernetes can be used anywhere. In fact, by abstracting the complexity of the underlying infrastructure, K8s can run on any type of cloud platform, on-premise data center or hybrid models, enabling easy migration between different cloud providers. This allows companies to run their applications wherever they need them and easily migrate to different platforms as needed. Therefore, Kubernetes enables organizations to focus on developing and delivering applications without worrying about the operational overhead, improving company reliability and reducing the time and resources attributed to daily operations. [9]

According to CNCF, Kubernetes is emerging as the 'operating system' of the cloud. The 2022 CNCF survey outlines how 64% of organizations are already adopting K8s as container orchestrators in production, while 25% are evaluating or piloting its adoption. [2]

2.3.1 K8s Features

Kubernetes comes with a lot of interesting features and benefits, that we can summarize as follow: [6]:

- *Scalability*: Kubernetes enables horizontal scaling of applications, allowing them to handle increasing workloads efficiently. Kubernetes autoscaling feature enables automatic spin-up of new containerized application instances as needed to handle the additional workload.
- *Flexibility*: Kubernetes supports various deployment strategies, including rolling updates, canary releases, and blue-green deployments, enabling organizations to iterate quickly and experiment with new features.

- *Portability*: Kubernetes abstracts away the underlying infrastructure, enabling applications to run consistently across different environments, from on-premises data centers to public and hybrid clouds.
- *Resilience*: Kubernetes provides built-in mechanisms for fault tolerance, self-healing, and automated failover, enhancing application reliability and availability.
- *Automation*: Kubernetes automates repetitive and manual tasks related to container management, such as deployment, scaling, service discovery, and load balancing, reducing operational overhead and improving efficiency.
- *Service Discovery and Load Balancing*: Kubernetes offers built-in service discovery and load balancing capabilities, enabling applications to communicate with each other seamlessly and distribute incoming traffic across multiple instances.
- *Rolling Updates and Rollbacks*: Kubernetes supports rolling updates and rollbacks of application deployments, enabling organizations to deploy new features or updates with minimal downtime and risk.
- *Security*: Kubernetes provides built-in security features, including network policies, role-based access control (RBAC), and container secrets management.
- *Storage Orchestration*: Kubernetes supports various storage options for containers, including persistent volumes and storage classes, for managing application data and stateful workloads.
- *IPv4/IPv6 Dual-Stack Provisioning*: Kubernetes enable automated allocation and management of IPv4 and IPv6 addresses to Pods and Services.

These benefits and features contribute to Kubernetes' popularity and success as a leading container orchestration platform in modern cloud-native environments.

The next section delves into the main features of Kubernetes, exploring the architectural components and resources useful for understanding the thesis work laid out in the next chapters.

As we will see, Kubernetes resources can be expressed via a YAML file, that is, a configuration file that adopts a JSON format. Then, using the **kubectl apply -f <filename>** command, you can provide that resource as input to the apiserver to be created and deployed within the cluster, if it doesn't exist yet.

2.4 Kubernetes Architecture and Resources

Before delving into the components of Kubernetes, it is essential to understand the fundamental elements that make up the Kubernetes architecture.

Each Kubernetes cluster consists of a set of worker machines, simply called *Worker Nodes* or *Nodes*, that run containerized applications, and every cluster has at least one worker node. Each worker node can be a virtual or physical machine that contains the services necessary to run containers.

A K8s cluster implements a master-slave architecture, consisting of a control plane (called *Master Node*) and one or more worker nodes (slaves) that actually run applications. Each component plays a crucial role in managing and orchestrating containerized workloads within a Kubernetes cluster.

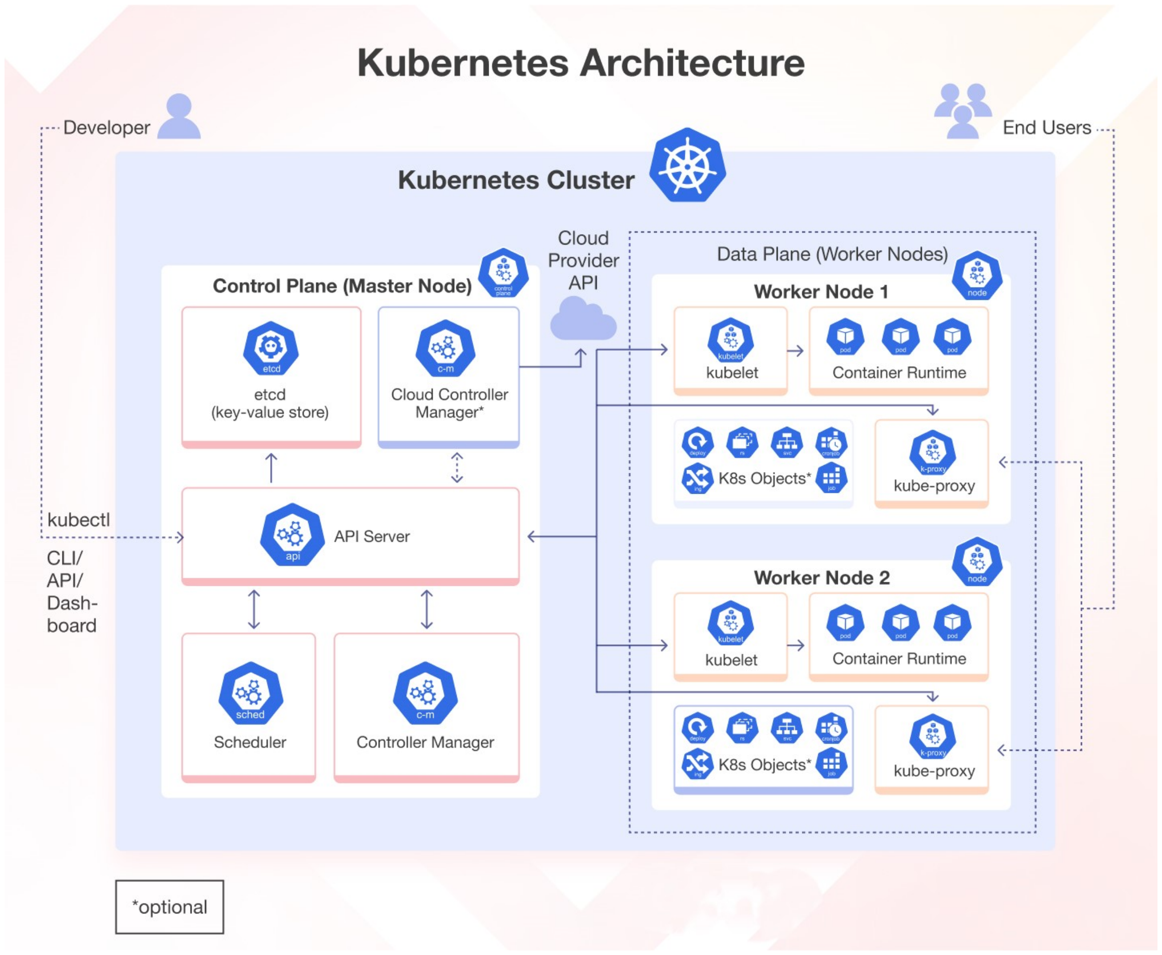


Figure 2.3: K8s Cluster Architecture (source: Simform)

The smallest unit that can be scheduled on the cluster is called *Pod*, and can have one or more containers inside. The worker nodes deploy, run, and manage Pods (i.e. the containerized applications within), while the control plane is responsible for managing the worker nodes and the Pods in the cluster, and it runs a scheduler service that automates when and where the containers are deployed based on developer-set deployment requirements and available computing capacity. Each worker node requires the installation of the tool that is being used to manage the containers — container runtime, such as Docker or Containerd — and a software agent called a Kubelet that receives and executes orders from the master node, interacting with the container runtime. [10][11]

2.4.1 Master Node

The Master Node serves as the brain of the Kubernetes cluster, responsible for managing and coordinating all activities within the cluster. The control plane's components make global decisions about the cluster (such as pod scheduling, for example), as well as detecting and responding to cluster events.

As shown in Figure 2.3 it is composed by several components: [10]:

- *Kube-apiserver*: The API server acts as the front-end for the Kubernetes control plane, exposing the K8s APIs for all administrative tasks and enabling communication with the cluster to configure and manage K8s resources.
- *Etcd*: As a distributed key-value store, etcd serves as the cluster's persistent storage for all Kubernetes cluster data, including configuration settings, state information, and metadata.
- *Kube-scheduler*: The scheduler component is responsible for assigning newly created pods to nodes within the cluster based on a series of context information like resource requirements, constraints, and availability.
- *Kube-controller-manager*: The controller manager oversees the operation of various controllers responsible for maintaining the desired state of the cluster. These controllers include the node controller, replication controller, endpoint controller, and service account controller, among others.
- *Cloud-controller-manager*: In cloud-based Kubernetes deployments, the cloud controller manager interacts with the underlying cloud provider's APIs to manage resources such as virtual machines, load balancers, and storage volumes. In our case, this was not used since we implemented an on-premises K8s cluster.

2.4.2 Worker Node

Nodes, also known as Worker Nodes, constitute the computing units of a Kubernetes cluster, where containerized applications run.

Each node hosts multiple components responsible for managing containers and providing essential cluster services:[10]:

- *Kubelet*: The kubelet agent runs on each node and is responsible for communicating with the control plane, managing pods on its behalf, and ensuring that containers are running within pods as expected.
- *Kube-proxy*: The proxy component enables network communication between pods and external clients by managing network routing, load balancing, and service discovery within the cluster. It maintains on nodes the network rules that are needed by the pods for network communication.
- *Container Runtime*: It is a fundamental component responsible for pulling container images, creating containers, and managing their lifecycle on each node within the Kubernetes environment. Kubernetes supports various container runtimes, including Docker, Containerd, and Cri-o. In our case, we adopted Containerd.

2.4.3 Pods

Kubernetes Pods are the smallest deployable units of computing that you can create and manage in K8s clusters. Essentially, a Pod represents a single Kubernetes resource encapsulating one or more tightly coupled containers that share storage, network resources, and a specification of how they have to be executed. Containers within the same Pod share the same network namespace (and thus share the same network stack and IP address) and can communicate with each other over the localhost interface, simplifying inter-container communication and facilitating collaborative workflows.

Consequently, pods can be viewed and managed as virtual machines or hosts, since they all have a unique IP address, and containers within pods can be treated as processes running within a virtual machine or host, since they run in the same network namespace and share an IP address.

Thus a pod can be conceptualized as an enclosure around a set of one or more containers, providing isolation from other independent pods running in the cluster. This encapsulation facilitates cohesion among the containers co-located within the Pod, allowing them to work together smoothly: whenever a pod running multiple containers is deployed, they are all created and executed correctly within it at all times. [12]

Pods in Kubernetes offer several key features that contribute to their usefulness and flexibility within the platform. First, Pods support horizontal scalability; that is, identical Pods can be replicated and deployed across the Kubernetes cluster to meet application demand. This scalability enables efficient resource utilization and high application availability. In addition, Kubernetes Pods incorporate robust self-healing mechanisms: if a Pod or its underlying node fails, Kubernetes automatically reschedules the Pod onto a healthy node, ensuring continuous operation of the application. This resilience is essential for maintaining application availability and reliability in dynamic environments.[12]

Typically, each pod will certainly run a main application within a primary container and possibly (depending on the needs of the application and cluster administrators) one or more sidecar containers as well.

Sidecar containers are a powerful concept in Kubernetes Pods, allowing the functionality of the main application to be augmented and extended without changing its code base. Sidecar containers run alongside the main container within the same Pod, sharing the same lifecycle and resources. This architecture facilitates the implementation of additional functionality such as logging, monitoring, security or proxying alongside the main application container. Sidecar containers allow for improving the resilience, observability and security of Kubernetes workloads by separating issues and promoting modular design patterns. In fact, by leveraging sidecar containers, developers can improve the capabilities of their applications without introducing complexity or tightly coupling different functionalities, thus promoting scalability, maintainability and agility of Kubernetes environments.[13]

As we will see in Chapters 5 and 6, the sidecar concept will be essential for implementing Service Mesh solutions in the cluster: the Istio sidecar container is injected within each pod at its boot, and is responsible for intercepting and managing pod network traffic produced or directed to the main application container (since containers within the same pod share the same network namespace, and thus network stack and IP address); instead, the main container executes only the application logic, depending on the received traffic that is passed by the sidecar container.

2.4.4 Deployments and ReplicaSet

Kubernetes Deployments are a cornerstone of managing containerized applications within Kubernetes clusters. Deployments provide a declarative way to define and manage the lifecycle of application instances, ensuring consistency and reliability across environments: users can specify the desired state of their applications, including the number of replicas, container images, and deployment strategies. Kubernetes then orchestrates the deployment resource, automatically

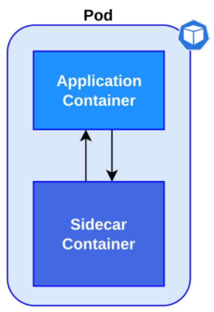


Figure 2.4: Kubernetes Pod With Sidecar (source: Kubebyexample)

managing the creation, scaling, and updating of application instances within pods to meet the specified requirements.

Deployments also offer robust features for rolling updates and rollback mechanisms, enabling seamless transitions between different versions of applications while maintaining availability and stability. Overall, Kubernetes deployments streamline the process of managing and scaling applications, promoting efficient and reliable operations in dynamic containerized environments. [14]

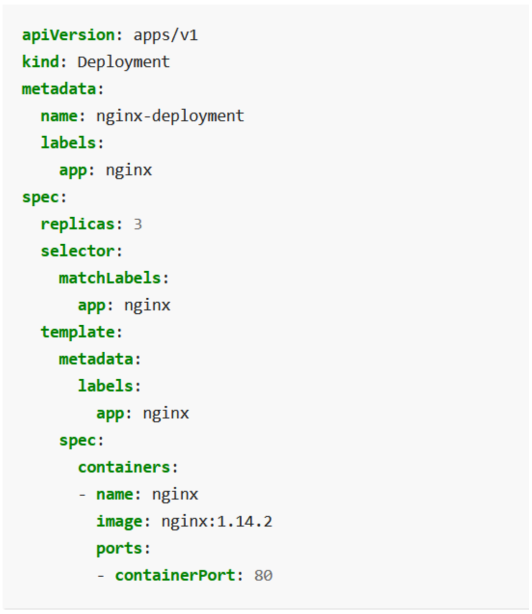
A ReplicaSet instead, is a resource whose purpose is to maintain a stable set of replica Pods running at any given time. Therefore, is adopted by K8s to guarantee the availability of a specified number of identical Pods.[15]

Figure 2.5 shows an example of a Deployment. Once we run the **kubectl apply -f <filename>** command, a Deployment named nginx-deployment is created, indicated by the *.metadata.name* field. The Deployment creates a ReplicaSet that creates three replicated Pods: whenever any of the 3 replicas terminate or has some issues, K8s immediately takes action to reestablish the desired state, that is, to ensure that there are always 3 replicas operating at any time. The *.spec.selector* field defines how the created ReplicaSet finds which Pods to manage. In this case, you select a label that is defined in the Pod template. The Pods are labeled app: nginxusing the *.metadata.labels* field. The *.template.spec* field, indicates that the Pods run one container, named nginx, which runs the nginx Docker Hub image at version 1.14.2 and uses the Pod's port 80 to expose the related service. Each pod can be scheduled anywhere within the cluster. [14]

2.4.5 Namespaces

Kubernetes namespaces provide a mechanism for logically partitioning and isolating resources (e.g. Deployments, etc.) within a cluster, ensuring the integrity of workloads. By creating distinct namespaces, Kubernetes users can segregate their applications, services and resources, establishing clear boundaries and access controls.

Each namespace functions as a virtual cluster within the larger Kubernetes environment, with its own set of K8s objects and policies. Users can define role-based access control (RBAC) policies at the namespace level, regulating who can interact with resources within that namespace. In addition, user namespaces facilitate resource quota management, allowing administrators to allocate and restrict resource usage for individual namespaces. Overall, Kubernetes user



```yaml
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:1.14.2
        ports:
        - containerPort: 80
```

Figure 2.5: An Example of a Deployment Resource (Source : Kubernetes Documentation)

namespaces promote multi-tenancy, security and resource isolation, facilitating collaboration and efficient management of different workloads within Kubernetes clusters.

When you create a Service resource, it creates a corresponding DNS entry. This entry is of the form *<service-name>.<namespace-name>.svc.cluster.local*, which means that if a container only uses <service-name>, it will resolve to the service which is local to a namespace. [16]

Namespaces cannot be nested inside one another: the various namespaces in the cluster co-exist isolated side by side on a single flat level, and each Kubernetes resource can only be in one namespace. Kubernetes starts with four initial namespaces: **default** (a namespace that Kubernetes uses by default for newly created resources, unless otherwise specified in the .metadata.namespace field of the YAML configuration file) , **kube-public**, **kube-node-lease**, and a namespace for objects created by the Kubernetes system named **kube-system**. However, in a production cluster the default namespace should not be used. Instead, it is suggested that other namespaces be created and used. [16]

2.4.6 Service Account and RBAC

A Service Account is an identity created inside the Kubernetes cluster, and it is used by pods to interact with the K8s API server and other cluster services in a secure manner. Each service account is associated with a specific namespace and provides a mechanism for authenticating and authorizing access to resources within the cluster.

Service Accounts are also associated with secrets, which are securely stored and mounted into pods as volumes. These secrets contain credentials or tokens that pods use to authenticate with external services or APIs. By leveraging Service Accounts and secrets, Kubernetes provides a secure mechanism for managing sensitive information and access credentials within the cluster.

Typically each pod is assigned a unique Service Account: whenever pods interact with the

Kubernetes API server, the associated token is used to authenticate requests. This mechanism ensures that only authorized pods can access cluster resources and perform actions within their namespace. [17]

Role-Based Access Control (RBAC) is a Kubernetes feature that governs access to cluster resources based on predefined roles and permissions. RBAC enables administrators to define granular access policies for the API server, specifying which actions a given Service Account can perform on specific resources. By assigning RBAC roles to Service Accounts, administrators can enforce security policies and control access to sensitive resources within the cluster.

Implementing RBAC and Service Accounts is crucial for securing cluster access and preventing unauthorized actions within the Kubernetes environment. [17]

2.5 Kubernetes Network Model

Kubernetes defines a network model that helps provide simplicity and consistency across a range of networking environments and network implementations. It defines how pods and services within Kubernetes communicate with each other, and how network traffic is routed and managed within the cluster. It provides some principles to be followed for implementing networking within a cluster [18][19]:

- Each pod is assigned a unique IP address
- Containers within the same pod share the same network stack, IP address and range of ports, and communicate via loopback
- Pods can communicate with all other pods in the cluster using the pod IP addresses (without performing any Network Address Translation)
- Isolation (restricting what each pod can communicate with) is defined using network policies

This model is implemented by the container runtime running on each node: it leverages the Container Network Interface (CNI) plugins to manage their network and security capabilities. CNI plugins are responsible for tasks such as assigning IP addresses to pods, setting up network interfaces, configuring routing, implementing network policies, and enabling communication between pods and external networks. There are many CNI plugins available, each with its own features and capabilities, allowing users to choose the one that best fits their requirements. [20].

However, the K8s network model also introduces several interesting features, reported in the next subsections.

2.5.1 Kubernetes Services

The Kubernetes workloads IPs are assigned when the pod that runs them is created, but they can change over time for various reasons, such as pod rescheduling, node failures, or horizontal scaling. Kubernetes Pods are, therefore, continuously created and destroyed to adapt to the desired state of the cluster. Since pods are ephemeral resources, it is difficult for other pods or external services to reliably communicate with pods using their IP addresses directly since they are unreliable and unstable. The Kubernetes network model introduces the concept of Service, with the goal of solving this problem.

The Services provide a way to abstract access to a group of pods as a network service by providing a stable virtual IP address (which can be discovered using Kubernetes DNS) that can be used to contact the pods hiding behind. Each service in Kubernetes automatically gets a DNS name that resolves to the cluster IP address of the service. This ensures that the DNS name and virtual IP address remain constant throughout the lifetime of the Service, even though the pods backing the Service may be continuously created or destroyed and the number of pods supporting the Service may change over time.

Services in Kubernetes also provide load balancing across multiple pods that belong to the service. This ensures that traffic is evenly distributed among the pods, improving the availability and reliability of applications. [21].

There are several types of services in Kubernetes, each of which has a specific purpose and meets different network requirements. Here are the most common types of services: [21]:

- *ClusterIP*: This is the default type of service used if you don't explicitly specify a type. It exposes the service on an IP address which is accessible only within the cluster.
- *NodePort*: This type of service exposes the service on each node's IP address at a static port. It allows external clients to access the service by connecting to any node in the cluster on the specified port. Traffic is then forwarded to the service within the cluster.
- *LoadBalancer*: This type of service provisions an external load balancer (like those provided by cloud providers) that routes traffic to the service.

2.5.2 Ingress and Ingress Controller

Ingress is another Kubernetes resource that enables external access to services within the cluster. It acts as a layer of abstraction for HTTP and HTTPS routing, allowing cluster external traffic to be routed to the appropriate services based on rules defined in the Ingress resource. The Ingress resource allows HTTP and HTTPS requests with particular domains or URLs to be mapped to some specific Kubernetes services, based on rules defined through that resource. But in order for the Ingress resource to work, the cluster must have an ingress controller running.

An Ingress controller is a Kubernetes component responsible for managing and configuring the underlying load balancer or reverse proxy that routes external traffic to services within the cluster based on the rules specified in the Ingress resources. The Ingress controller watches for changes to Ingress resources in the Kubernetes API server and reconfigures the load balancer or reverse proxy accordingly.

There are several Ingress controllers for Kubernetes, each with its own features and capabilities. In this thesis work, since Istio Service Mesh was being analyzed, it was decided to adopt its built-in Ingress Gateway. [22][23]

2.5.3 Calico CNI Plug-in Networking

Calico implements the Kubernetes CNI as a plug-in and provides agents for Kubernetes to provide networking for containers and pods. It creates a flat Layer-3 network by dividing an initial CIDR into a series of smaller IP subnets (address blocks) and assigning one or more of these blocks to nodes in the cluster. It then assigns a fully routable IP address to each pod, depending on which node it is scheduled on (and thus on which logical subnet). When routing pod traffic, Calico uses the node's local route tables and iptables: all pod traffic goes through iptables rules before being routed to its destination. A pod has its own network stack that is completely isolated from the others, leveraging Linux network namespaces.

Each pod is connected to the internal host's virtual network (i.e. host network namespace) using a pair of virtual Ethernet interfaces (called a veth pair). A pod sees eth0 as its local interface, and the host kernel generates an interface name for each pod that begins with "cali." The Calico plugin then sets up the host's network namespaces to act as a virtual router/switch: each pod residing on the host is connected to that virtual router, and Calico makes sure that the virtual router knows where all the pods in the rest of the cluster nodes are programmed so that it can forward traffic to the right places. All traffic forwarding occurs natively within the Linux kernel.

Traffic between pods on the same node is routed locally and traffic between pods on different nodes is routed over the underlying network: in some cases, the underlying network does not know how to forward the pod traffic, and therefore it is needed to run an overlay network.

Calico supports two overlay modes, VXLAN and IP-in-IP, which are implemented by virtual interfaces within the Linux kernel. Whenever a pod sends a packet to a pod on a different node, it is encapsulated using VXLAN or IP-in-IP in an external packet that uses the node's IP addresses and hides the pod IPs of the original internal packet. This creates a kind of tunnel between two nodes. The underlying network handles this packet like any other node-to-node traffic. On the receiving node, the packet is decapsulated to extract the original packet and then delivered to the correct destination pod using the host's internal virtual network. [24][25]

By default, the Kubernetes CNI plugin does not restrict any network traffic between pods, therefore, any pod can communicate with any other pod in the cluster, as well as with external sources. This can pose security risks, as malicious pods can exploit vulnerabilities in other pods or access sensitive data.

Chapter 3

Security Challenges And Zero Trust Model

3.1 Limitations of the Castle and Moat Approach

Until a few years ago, the traditional approach to cybersecurity taken by companies and datacenters to defend their networks was centred on the idea of defining a perimeter within which their services, data and devices resided, building strong boundary defence measures to protect everything within the organisation's network from outsider threats, and implicitly trusting everything and everyone operating within that perimeter.

Also known as the 'castle and moat' approach, the idea behind this model is to imagine the corporate network to be protected as a castle and the network perimeter surrounding it as a moat, creating a clear separation between the internal trusted zone and everything outside it, namely the external untrusted network (i.e. the Internet).

Typically, organizations deploy many resources such as Firewalls, Intrusion Detection Systems and Intrusion Prevention Systems, which are useful for implementing security at the network perimeter, detecting external attackers and possibly blocking their penetration into the network.

These types of systems are suitably configured to act as a separation as well as the only point of contact between the trusted network to be protected and the rest outside, thus forcing any north-south traffic (i.e. that which flows from outside to the organization's network, and vice versa) to necessarily pass through these security solutions, being able to inspect such traffic and detect any threats or unauthorized traffic and take timely countermeasures.

Unfortunately, although these solutions help to protect against external threats, they are not as effective in detecting and blocking insider threats and attacks, or data breaches. In fact, this perimeter-oriented security model does not take into account the possibility that such perimeter security measures may sometimes fail or be unable to detect malicious traffic, due in part to evolving threats and the ever-increasing skill of cybercriminals. In addition, malicious actors or employees could infiltrate and operate from within the corporate network.

The lack of preventive security solutions inside the perimeter that would allow visibility into activities within the network, and to inspect, authenticate and authorize east-west traffic is the biggest security flaw of the castle-and-moat approach.

In fact, for most organizations, east-west traffic constitutes the majority of data center and

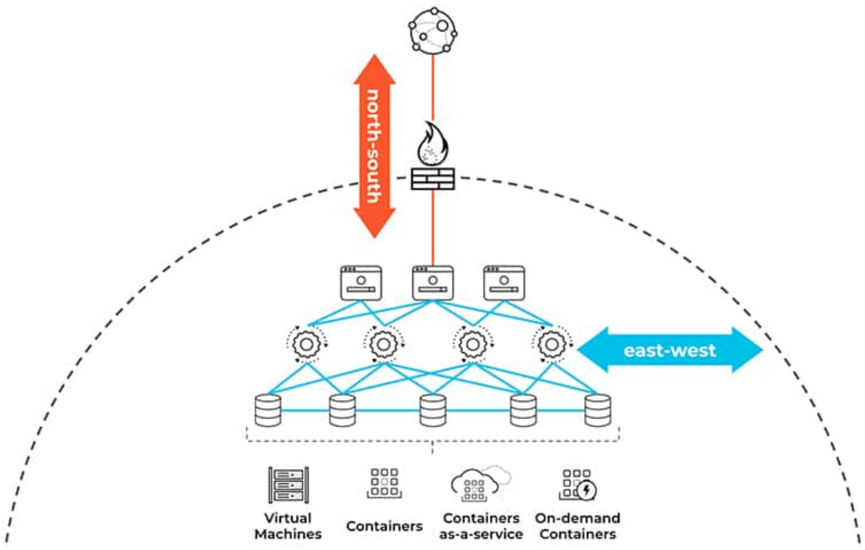


Figure 3.1: Corporate North-South and East-West traffic (source: Paloalto Networks)

cloud communication flows, and perimeter defenses have no visibility into this internal traffic.

As a consequence, the impact of a security incident and any lateral movement within the network is not mitigated in any way.

If an attacker gains access to the network, i.e. if he is able to overcome the security measures in place at the border, then he could move freely sideways and gain access to any sensitive service or system within the castle, perhaps manipulating or worse, exfiltrating sensitive data, or performing other types of attacks such as ransomware or cryptojacking.

Moreover, as a result of the ongoing digital transformation, there is no longer an easily identifiable corporate network edge to protect. The typical enterprise infrastructures where services and data reside, are becoming increasingly complex, often consisting of a combination of multiple internal networks with a large number of endpoints, remote offices, IoT and mobile devices, and cloud services. Relying upon IaaS, PaaS, and SaaS services, organizations are migrating their infrastructure, platforms, and applications (or even part of them) to cloud. Therefore modern enterprise digital infrastructures are no longer confined within a single perimeter that clearly separates them from the rest, but span on-premises data centers as well as private, public and hybrid clouds. The Covid pandemic added further complexity to the landscape by introducing remote working, which remains widely used by companies to this day.

Due to the overall complexity of the scenario just described, traditional perimeter-based network security solutions alone are considered obsolete and provide poor granularity of access control, as there is no unique and easily identifiable perimeter for the enterprise, as well as proven insufficient, as once attackers breach the perimeter, further lateral movements are not prevented. [26]. To address this complexity and the new challenges facing cybersecurity, a new model called Zero Trust was introduced.

3.2 Zero Trust Security Model

The inadequacy of the 'castle and moat' approach and the difficulties in defining the perimeter of an organisation's information systems had already been highlighted in 2003 by the Jericho Forum and later taken up by John Kindervag, a researcher at Forrester, who, in 2010 proposed as a solution a more rigorous approach to cybersecurity and access control within companies, called Zero Trust.

Zero Trust is a strategic approach to cybersecurity that secures an organization by eliminating the concept of a trusted corporate network and the implicit trust of any entity within it by continuously analyzing, verifying, and authorizing every request and interaction with any IT resource before it is allowed. The term ZT refers to a paradigm encompassing a set of concepts and principles whose goal is to prevent unauthorized access to data, services and devices whenever your internal network is compromised, leveraging upon the motto "never trust, always verify".

Designing and implementing the Zero Trust Model for a given company means assuming that threats might already be inside the network, perhaps through a malicious insider or an attacker who has breached the perimeter defense, and thus considering any corporate network no different or more trustworthy than any other nonenterprise, untrusted network.

An operative definition of zero trust and zero trust architecture, provided by NIST in the 800-207 standard, is as follows:

"Zero trust (ZT) 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. Zero trust architecture (ZTA) is an enterprise's cybersecurity plan that utilizes zero trust concepts and encompasses component relationships, workflow planning, and access policies. Therefore, a zero trust enterprise is the network infrastructure (physical and virtual) and operational policies that are in place for an enterprise as a product of a zero trust architecture plan." [26]

When organizations require Zero Trust and Zero Trust architecture, the starting point is assuming that all users, devices, and applications within the enterprise network can be potentially compromised and should not be trusted by default. Therefore, there is no entity within the corporate infrastructure that can be considered a secure origin and whose operations are implicitly considered trusted without performing any kind of access control.

In contrast, in order to mitigate uncertainties (since they cannot be completely eradicated), continuous verification of the identity and authorization of all requesting entities is required before granting access to an IT resource, shrinking implicit trust zones and making the application of access control as granular as possible. At the same time, it is also important to maintain resource availability and minimize temporal delays in authentication mechanisms.

However, adopting a Zero-Trust strategy does not involve abandoning the castle-and-moat defense. On the contrary, it means implementing the principle of defense-in-depth by integrating firewalls and IDS/IPS with other types of countermeasures. Implementation of ZT within your network offers an additional layer of security that complements the perimeter one, based on the assumption that the enemy is not only at the gates but may already be inside the perimeter, thus enabling organizations to be prepared for the eventuality and to respond more effectively and timely to emerging threats, both external and internal.

The picture just above (Figure 3.2) shows the difference between a traditional trust-based network and a zero-trust network.

By looking at the traditional network scheme, i.e. a castle-and-moat approach, it is clear how the whole enterprise network is considered an implicit trust zone. Therefore once the attacker

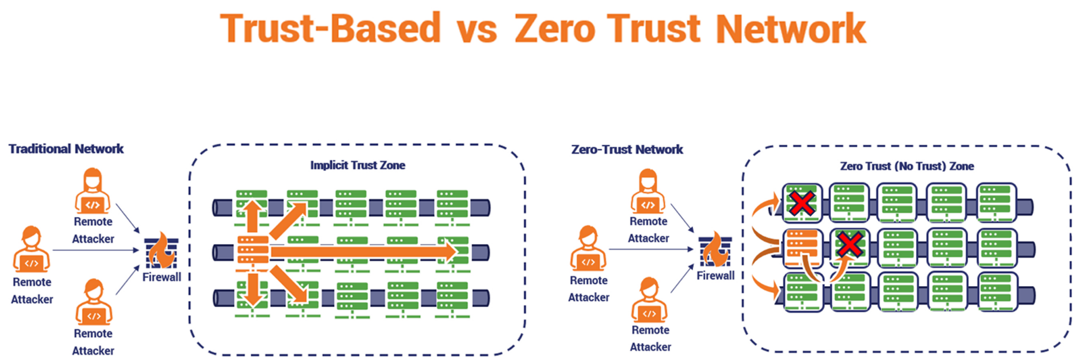


Figure 3.2: Traditional Network vs Zero Trust Network (source: The SSL Store)

overcomes the first defense line (FW and maybe also IDS/IPS, which are not shown here), it is free to move far and wide throughout the organization's network without any impediment or mitigation, since any action taken once inside the network is implicitly considered trusted and safe.

Therefore, it may be sufficient to bypass the firewall and compromise any corporate resource, and then carry out an attack from within the private network, maybe reaching further sensitive targets.

The traditional model alone no longer works when techniques like credential phishing and session hijacking are carried out. More robust security and authentication measures are needed.

The zero-trust environment, on the other hand, differs from a traditional security approach in that zero-trust implies the need to continually prove trustworthiness, even once the perimeter and related security controls have been passed.

It is clear from the image how a firewall (or other perimeter-based security measures) is still present even when Zero Trust is implemented. ZT architecture doesn't involve removing perimeter defense, instead, it implies integrating this kind of solution with additional security measures that enable individual resource protection even when the first defense line fails. In fact, the image on the right shows how even once the attacker gains access to the network and the resource is compromised. It will be allowed access only to the business assets it needs to complete certain tasks and thus, for which it is actually authorized. Any other type of request to other resources on the network (and thus unauthorized lateral movement) will be denied, obtaining threat mitigation.

3.3 Zero Trust Network Principles

In 2020, the U.S. National Institute of Standards and Technology (NIST) developed and published NIST SP 800-207, a document that addresses the Zero Trust approach in-depth, providing a comprehensive but detailed overview of the fundamental tenets, architectural components, and best practices for designing and implementing a Zero Trust architecture.

According to the 800.207 NIST pubblication[26] , we can summarize the key assumptions and principles to consider when wanting to design and implement a Zero Trust network as follow:

1. *All IT assets, data sources, devices, and computing services within the corporate network are considered resources and can be compromised and subverted at any time.*

2. *The entire enterprise network infrastructure is not considered an implicit trust zone, and therefore all resources must act at any time as if an attacker is already within the private network.* This means that all communication that involves an enterprise resource, whether incoming or outgoing, should be done in the most secure manner available independently of network location, always protecting message confidentiality, authentication and integrity, and providing source authentication.

3. *'Never Trust, Always Verify'. Every request for access to any enterprise-owned asset must always undergo a rigorous and dynamic security posture evaluation process, as no source requesting any resource should be inherently trusted.* A Policy Enforcement Point must be placed in front of any enterprise resource and, before granting access to it, must be able to perform source identity verification and request authorization for any request, regardless of whether it comes from an external or enterprise-owned source. This means that communications and access requests coming from entities located in the corporate network infrastructure should not be considered implicitly trusted. They must be treated as any other requests coming from non-enterprise networks and therefore must undergo the same security requirements for resource access.

4. *Every entity (users, services and devices) that interacts with corporate resources must possess a reliable and verifiable digital identity, even if it is internal to the enterprise network.* Strong and verifiable authentication mechanisms must be continuously performed for every access request received from any corporate resource, in order to demonstrate the digital identity of the source (and thus that it is not an imposter) and then be able to retrieve and verify the corresponding authorizations consecutively.

5. *Access to every single enterprise resource should be granted on a per-session basis, ensuring least-privilege access and default deny approach.* When requesting access to any corporate resource, a least-privilege policy should be applied by providing the requestor only the minimum amount of privileges required to complete a given task. By default any request is denied and requires a separate evaluation and authorization process that enables it: once the request of a given entity is granted, it does not automatically grant it access to other different resources, or to the same resource in the future, but further requests for access must be submitted and evaluated separately.

6. *Ensure visibility of network operations at any time, and continuous inspection and monitoring of activities.* Since no corporate resource is inherently trusted, an enterprise should continuously collect data and logs about asset security posture, network traffic and access requests, process that data, and try to detect and analyse suspicious operations that don't reflect the "normal" behaviours. Constant monitoring of network activity, as well as traffic patterns and request flow, can help to gain complete visibility into the activities taking place within the network, and possibly detect anomalous and/or malicious activity and promptly take any useful mitigation to limit the impact of a security incident. In addition, the insights gained from this data can be used to improve policy creation and enforcement and thus the security posture for any resource within the network.

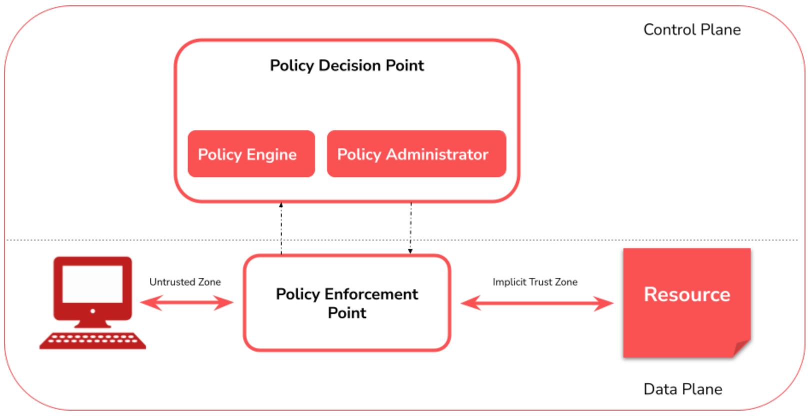


Figure 3.3: A Conceptual Model of Zero Trust Logical Components (source: B-Nova)

7. *Assets and workflows moving between enterprise and nonenterprise infrastructure should have a consistent security policy and posture.* Often nowadays some corporate resources do not reside on the company-owned network, and in addition, they may need to use some non-company network services (e.g., DNS resolution): some examples are remote users (i.e. those connecting from remote nonenterprise networks) or workloads migrating from on-premises data centers to nonenterprise cloud instances. It is extremely important to retain assets and workloads security posture when moving to or from enterprise-owned infrastructure, and thus a reliable PEP can be securely deployed to protect resources located on a nonenterprise infrastructure as well.

In the next chapters, we will consider and follow all these basic principles and guidelines when defining a Zero Trust architecture for our Proof Concept.

3.4 Zero Trust Architecture And Microsegmentation

Whenever implementing a zero-trust architecture, it should always be considered that, as also reported by NIST [26], , any policy-based access control system consists of 2 main components: a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP). Actually, PDP is broken down into two logical components: Policy Engine (PE) and Policy Administrator (PA).

The main idea is that PDP and PEP are those components that work together to apply a set of controls, performing authentication and authorization processes, and thus determining whether or not access to a given resource can be granted to a specific entity.

- *PEP*: The Policy Enforcement Point is the element that acts as a gateway. It is placed in front of assets to be protected, and is responsible for enforcing authentication and authorization mechanisms, enabling, monitoring, and terminating connections between a subject and an enterprise resource. It communicates with the PA to forward requests and/or receive policy updates to be applied from the PDP.[26].

- *PE*: The Policy Engine is the part of the PDP responsible for deciding which entity is authorized to access each specific resource, by defining access control policies. The PE is strictly coupled with the PA component. The policy engine typically makes and logs the decision (as approved, or denied), and the policy administrator executes the decision.[26].

- *PA*: The Policy Administrator component is responsible for establishing and/or closing the communication path between a subject and a resource, by communicating with the PEP. It relies on the access control policies and decisions provided by the PE to configure the PEP to deny or allow resource access requests coming from a given subject in the untrusted zone.[26].

There are also some implementations that treat the PE and the PA as a single service, commonly named PDP. Note in Figure 3.3 that PEP represents a clear separation between what is considered an implicit trust zone and the untrusted one.

By following Zero trust principles and concepts, PDP/PEPs should be moved and maintained as close to the resource as possible. This should be done to reduce the implicit trust zone to a minimum, and explicitly authenticate and authorize all enterprise subjects, assets and workflows that want to access the asset. This also permits the implementation of granular access rules and to enforce the least privileges needed to perform the action in the request.

However, there are multiple ways through which an organization can establish a Zero Trust Architecture for its workflows, still considering and adopting the general components and architecture described above. These methods differ in the components adopted and the primary source of policy rules within the organization.

Chapter 3.1 of the NIST SP 800-207[26] outlines 3 possible different approaches, focusing on Identity-based, Network-based, and Microsegmentation-based strategies.

Every approach adheres to all the principles of ZT (outlined in section 3.3 of the current chapter), but may prioritize one or two of them as the primary driver of security policies.

1. *Identity-based Approach*: This approach focuses on verifying user identities and implementing access controls based on user attributes like role, location, and device posture. It emphasizes robust authentication methods and precise access controls to ensure only authorized users access resources.

2. *Network-based Approach*: The second approach leverages network infrastructure and Software Defined Perimeter technologies to implement Zero Trust. This approach involves creating secure perimeters around resources using software-defined policies rather than relying solely on traditional network boundaries.

3. *Microsegmentation-based Approach*: The last approach involves placing individual resources (or groups of them) on a single network segment protected by a security component that acts as a gateway. It extends the concept of network segmentation to the application or workload level, allowing for granular access controls and isolation of critical resources. The main idea behind is containing and limiting the spread of potential threats by properly isolating the workloads of individual applications.

A comprehensive zero-trust solution can include elements from all three approaches.

Each method has its strengths and weaknesses, and organizations may choose to adopt a combination of these approaches based on their specific security requirements and infrastructure.

As reported in the next chapters, I mainly focused on the microsegmentation approach, since, also analyzing the security issues and solutions on the market, it seemed to be the one best suited to implement Zero Trust within dynamic environments like Kubernetes clusters.

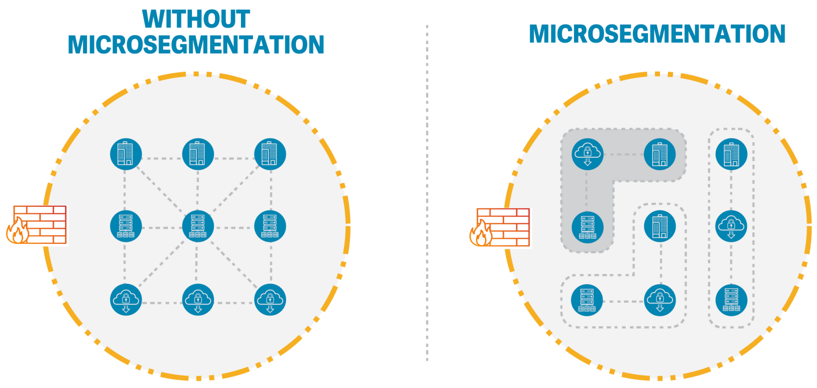


Figure 3.4: A Conceptual View of Networks With and Without Microsegmentation (source: Share Vault)

In cloud-native architectures such as Kubernetes, pods and containers inside are frequently spin up and down in seconds in order to guarantee workloads self-healing and scaling.

Because of the dynamicity of such environments, the IP addresses assigned to workloads running on the cluster are ephemeral, making it impossible to manage IP-based rules. With micro-segmentation instead, security policies are expressed in terms of identities or attributes rather than only on network properties such as IP address, protocol and port.

Microsegmentation might be enforced across application environments made up of microservices, which often have a lot of dependencies. In fact, in a microservice-based application, typically, each single microservice represents a different application workload. With micro-segmentation it is possible to create more flexible solutions because you can microsegment individual workloads and then apply access control policies specifically designed to keep them safe.

The goal is to isolate workloads (or groups of them) in a network to limit the effect and spread of malicious threats with policies that precisely control traffic in and out of specific workloads.

Each pod (or set of pods if the microservice consists of many replicas) running in the cluster will represent a different segment, shrinking the implicit trust zone around the single workload. Every workload will have its own security perimeter around it, thus trying to prevent lateral movement when the threat attempts to move from one segment/workload to another.

Figure 3.4 highlights how, without microsegmentation, every resource can be contacted by any other within the same perimeter. Each node represents a different workload operating on the enterprise network infrastructure. The dashed links highlight which resource can be reached by each individual node. Once sensitive data sources and critical assets and services have been found, we establish logical boundaries (represented in the figure above as dashed lines grouping resources) between business resources and workloads by adopting PDP/PEP components.

The goal is to allow each workload to contact and cooperate only with those resources strictly necessary to perform certain business activities. Any other communication or requests involving a pair of workloads that were not intended to interact for any reason will be automatically blocked and cannot leave the logical boundaries.

In this way, whenever a particular resource within the network has been compromised, in a worst-case scenario, the attacker would be able to affect only those resources with which the workload has been authorized to interact, greatly containing the spread and impact of threats within the network and protecting other critical resources that reside within distinct logical perimeters.

Therefore, I moved towards identifying relationships and dependencies among microservices deployed within the clusters, and implementing gateways (PEPs) around each different pod (hosting a given service or workload) dynamically grant access to individual requests from a client, asset or service. As already mentioned in section 3.3, identity is a crucial point in zero trust. In this case, the identity-based approach was also partially considered and implemented to achieve Proof of Concept, but the main emphasis was not on identities, as I did not dispose of a sufficiently robust IAM framework and authentication mechanisms to test and integrate into the ZT architecture.

3.5 Kubernetes Security Challenges

Thanks to its flexibility, scalability and automation properties, Kubernetes is now becoming increasingly popular and widely adopted by companies to deploy, run and manage their services. On the other hand, however, these modern environments and the related technologies adopted raise many security issues.

Compared to traditional networks, Kubernetes significantly expands the attack surface: as more pods and containers run on each node of the cluster and communicate with each other and with external services and networks, there are more entry points for attackers to exploit. In the case of a pod/container breach, the attack surface is directly related to the extent of the affected pods's communication with other pods and services. Furthermore, vulnerabilities contained within individual container images and in the CI/CD process of workloads on the cluster, combined with possible misconfigurations of cluster resources, could easily lead to privilege escalation and the creation of malicious pods in the cluster that would act unimpeded.

Then, since Kubernetes is a highly dynamic environment where pods and containers are constantly being created, terminated, moved across different nodes, and in general dynamically scaled up and down according to needs that are constantly mutating, it makes it challenging to maintain a consistent security posture for each of them.

Pods and containers need to communicate with other pods in the cluster and with other external endpoints. The number of interactions required by each workload with other services is often high, especially in a microservice-based architecture. But in an environment where pods are ephemeral entities continuously spin up and down, and a different IP address is assigned each time, it is difficult to implement network segmentation within the cluster due to the dynamic nature of the actors (and their network configuration) within the network and thus the complexity of manually configuring this type of policy. Moreover, manually setting up a security control for the various pods and managing their security configuration becomes impossible.

But what is even more critical is the lack of visibility into the pod operations within a Kubernetes cluster. As more and more containers are deployed, it becomes difficult to maintain adequate visibility into the components of this cloud-native infrastructure. Moreover, the distributed nature of containerized applications makes it difficult to investigate containers and pods that could pose a significant vulnerability or risk to the business quickly and detect anomalous behaviour.

Further complexity is introduced by the fact that the pods in Kubernetes often communicate using overlay networks, which create a distributed network that can be easily ported to any infrastructure. However, although flexible and powerful, this network model also reduces visibility and makes it difficult to regulate traffic using traditional tools such as firewalls, IDS and IPS.

Traditional network access controls operate at the host level and are unaware of the containerised resources running within a host: whenever two pods residing on different cluster nodes interact, due to the fact that pod-to-pod traffic is often encapsulated using IP-IP or VXLAN tunnels, these network solutions are unable to inspect and correctly understand the actual traffic exchanged. The same also applies to traffic entering and leaving clusters, as this traffic is usually netted by the host's iptables before entering or leaving the host. Finally, of course, these tools have no visibility into communications between pods deployed on the same nodes, since the related traffic never leaves the host. Thus, also traditional perimeter-based security approaches are useless when operating Kubernetes clusters, leaving also the north-south traffic exposed to security risks.

It is clear how Kubernetes network and security differs from traditional IT and infrastructure systems: many new aspects open the door to possible new threats and attacks, easily leaving room for bad actors to infiltrate and move laterally through the network and sensitive workloads if appropriate security measures are not taken.

Consequently, security in Kubernetes requires a paradigm shift from traditional network security due to its distributed, dynamic and container-centric nature. Therefore, the implementation of Zero Trust principles is essential (and even more important) in Kubernetes environments. But at the same time, it can be challenging, as the high level of dynamism of its component and the virtual network model introduced by the orchestrator do not allow visibility into traffic entering and leaving the pods and the cluster, and therefore it is not possible to adopt traditional network security solutions to implement micro-segmentation between workloads. It is necessary to ensure that a PEP is automatically placed in front of each newly created pod, regardless of whether the K8s platform (or the specific cluster node) resides in an on-premises datacenter or in a cloud provider's infrastructure, and that authentication and authorisation mechanisms for requests and communications between services are applied to any workload created within the cluster, regardless of the underlying infrastructure. PEPs must also enforce policies by tracking dynamic changes to cluster resources, updating them automatically (after explicit configuration by the system administrator) when new Kubernetes Deployment, services and workload replicas are created.

Chapter 4

Analysis Of Kubernetes Network Security Solutions

4.1 Istio Service Mesh

The advent of microservices architecture has led to a major increase in highly dynamic and distributed systems. But while microservices offer scalability, resilience and agility, they also introduce some issues regarding communication between services. As the number of services grows, discovering and locating services dynamically becomes complex, as well as ensuring security and compliance across distributed systems.

Gathering information about the behavior and performance of microservices is essential for effective system management and optimization, as well as from a security perspective. However, also monitoring a distributed system composed of numerous microservices raises significant observability issues.

Finally, because of their distributed nature, microservices are inherently susceptible to failures and latency issues, and this impacts communications and performance. But in the context of the microservice approach, service-to-service communications are what makes it possible to implement this architecture and achieve its benefits. Therefore, properly managing services communication and addressing these challenges becomes crucial.

A service mesh is a dedicated infrastructure layer responsible for managing and facilitate communication between microservices within a distributed application, enabling fast, efficient deployment and configuration. It consists of configurable, lightweight network proxies, called sidecars, which are deployed alongside each service instance, handling inter-service communication aspects such as traffic management, security, observability, and other cross-cutting aspects.

Each Service Mesh has 2 main logical components: the Data Plane (i.e. proxies that manage network traffic) and the Control Plane, which receives all the configuration resources and then pushes them to the Data Plane.

Without a service mesh, each microservice would be required to implement the logic governing inter-service communication in its code, which means that developers are less focused on business objectives. In addition, communication failures are more difficult to diagnose and troubleshoot because the logic governing inter-service communication is hidden within each service and there is no visibility into the behavior of microservices.

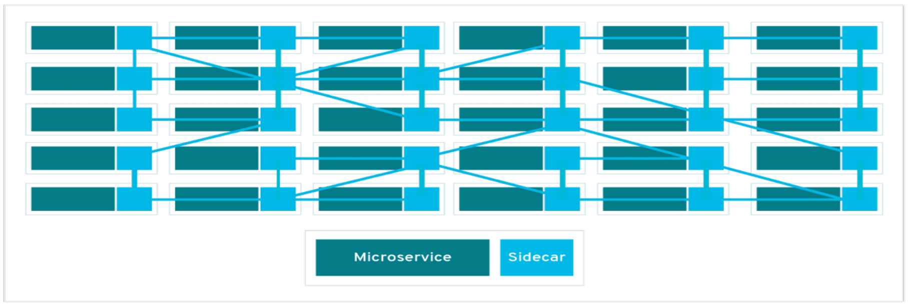


Figure 4.1: A Conceptual View of a Service Mesh (Source: Red Hat)

In Kuberneters, these network proxies are typically deployed as sidecar containers within K8s pods, decoupling the main business logic, implemented by the main container, from the communication functionalities implemented by the sidecar. These sidecars are configured to intercept traffic flows between services (both inbound and outbound), and properly manage and route requests on behalf of a Control Plane. In essence, a service mesh abstracts the complexities of network communication from individual services implementation, allowing developers to focus on business logic rather than on network configurations and communications management. [27]

In general, we can summarize Service Mesh capabilities as follow [28]:

- *Traffic Management*: Service Mesh provides advanced traffic management features such as load balancing, routing and traffic shaping. This allows cluster administrators to implement sophisticated traffic management policies without changing the application code.
- *Resilience and Fault Tolerance*: Service Mesh improve the fault tolerance of microservices communications by implementing resilience patterns such as circuit interruption, retries, and timeouts.
- *Observability and Monitoring*: Service mesh enhances observability by providing in-depth metrics, logs, and distributed traces. This allows administrators to monitor service interactions, identify bottlenecks, and troubleshoot issues more effectively.
- *Security*: Service Mesh tries to address some security issues by providing features such as mutual authentication, encryption, and fine-grained access control. This ensures that communication between services is properly protected, trying to prevent unauthorized access or tampering.

The security capabilities offered by service meshes and the ease and speed with which they can be configured without changing the source code of individual microservices is what led to the consideration of such solutions for implementing a zero-trust network.

Nowadays, there are several Service Mesh solutions on the market, most of them open-source. They all provide similar functionality, and differ in their implementation of the proxy and some extra or specific features. Each solution has its strengths and may be better suited to different use cases or environments. When choosing a service mesh solution, factors such as ease of use, performance, scalability, type of traffic that can be intercepted, and integration with existing infrastructure should be considered.

Of course, the discriminating factor adopted to choose the best one to realize this Proof of Concept was the security and observability capabilities provided by the various solutions considered, and Istio Service Mesh was the chosen one. Istio integrates tightly with Kubernetes, the de facto standard for container orchestration. This integration simplifies deployment and management for organizations already using Kubernetes for containerized workloads. As far as this thesis work is concerned, I have focused on the security features, so I will not go into further detail on the other features of Istio since they involve a lot of useless details.

4.1.1 Istio Architecture

From an high level architectural point of view, at its core Istio is composed of two main logical actors [29]:

- A **data plane**, which consists in a series of Envoy proxies (developed in C++) deployed within each microservice pod as a sidecar container. They are properly configured at the startup of each pod to intercept any traffic entering or leaving the main container running the application logic, and consequently be able to manage and control network communications between the microservices.
- A centralized **control plane**, which is in charge of configuring the data plane proxies to enforce authentication and authorization policies and collect telemetry data. It takes the desired configuration provided to the API gateway, and dynamically programs the proxies, updating them as the policies or the environment changes.

Typically Envoy intercepts pod's traffic using iptables rules, which are configured within each Kubernetes pod. When a Kubernetes pod is created or restarted, the Envoy sidecar container is initialized alongside the main application container within the same pod. Leveraging the fact that multiple containers within the same pod share the same network stack, IP address and ports, Istio Control Plane injects iptables rules into the network namespace of the pod, redirecting all incoming and outgoing traffic to the Envoy proxy running as a sidecar. These iptables rules ensure that traffic destined to the microservice's IP address and port is intercepted and forwarded to Envoy before reaching the main application container. [30]. Unfortunately, Envoy is only able to intercept TCP traffic, letting UDP and ICMP traffic flow freely. It supports gRPC, HTTP, HTTPS and HTTP/2 natively, as well as any plain TCP protocols.

Istio Control Plane provides service discovery, configuration and certificate management. It is named **Istiod**, and consists of 3 main components [29]:

- *Pilot*: Pilot provides service discovery for Envoy sidecar proxies, traffic management capabilities, and resiliency features like timeouts, retries, and circuit breakers.
- *Citadel*: Citadel is Istio's identity and certificate management component, responsible for issuing and rotating certificates for service-to-service communication within the mesh.
- *Galley*: Galley centralizes configuration validation, ingestion, processing, and distribution across Istio's control plane components.

4.1.2 Security Capabilities

Istio provides strong identity and robust management, powerful policy, transparent TLS encryption, and authentication, authorization and audit (AAA) tools to protect your services and data.

These security features enable it to be well suited to the requirements of a zero-trust network, ensuring security without requiring changes to the application code or infrastructure, following the *Security by Default* principle. It also allows for easy implementation of the *Defense in Depth* principle, as this Mesh service can be easily integrated with pre-existing security systems to establish multiple layers of protection. [31].

Here is a more detailed list of the features offered [31]:

- **Secure Services Communication**: Envoy can act as both TLS termination for incoming connections, as well as TLS origination when making connections to upstream clusters. TLS (Transport Layer Security) is a security protocol that enables the establishment of a secure communication channel and provides many properties to service traffic, such as peer authentication (for both client and server) before the channel is opened, confidentiality, authentication and message integrity, and protection against replay and filtering attacks. This means that proxy functionality can be leveraged to encrypt TCP traffic between services using mutual TLS (mTLS), ensuring that all data exchanged within the network is encrypted in transit, and thus achieving protection against eavesdropping, tampering, and several kinds of man-in-the-middle attacks. Even if the main application container produces plain traffic, this will be intercepted by the proxy through iptables rules and encrypted before leaving the pod. Viceversa, the incoming traffic will be first decrypted by the server-side proxy, and then sent as plaintext to the main container within the pod.
- **Identity and Authentication**: Istio assigns unique identities to each service within the mesh, typically represented by Service Accounts (acting as Kubernetes Service Identities). These workload identities are associated with certificates issued by the Certificate Authority (CA) built into Istio, ensuring their validity through the Public Key Infrastructure. When services communicate via mTLS, they present their certificates to each other for mutual peer authentication, ensuring that only authenticated and trusted services can communicate within the network. This subsequently allows communications between services to be authorized or denied based on their identities. Istio also allows client end-user authentication of each individual HTTP request (in addition to the peer client authentication), leveraging JWT tokens within the HTTP traffic: hardcoded identities and claims within the token can be extracted once the token is validated and used to authenticate the individual user or entity. Thus, two types of Authentication can be achieved through Peer Authentication and Request Authentication.
- **Authorization Policies**: Server-side Envoy proxies are able to apply authorization policies to intercept traffic entering the pod, controlling who can access what services within the mesh. Istio's authorization functionalities provide mesh, namespace and workload-level access control for workloads in the network. Policies can be defined based on attributes like service identity (extracted during mTLS authentication), request path, or source IP address. Fine-grained access control policies can be defined using RBAC. This enables administrators to specify which services can access other services and what actions they can perform. Istio enables the implementation of not only service-to-service authorization policies, but also user-to-service policies by exploiting the JWT tokens contained in HTTP traffic headers and the identities and claims extracted from them.
- **Observability and Telemetry**: As traffic passes through it, Envoy extracts rich telemetry data for all service traffic within a cluster, including cluster ingress and egress traffic, which can later be sent to monitoring systems to provide information on the behavior of the entire mesh. Administrators can monitor security-related metrics, logs, and traces to gain insight into communication between services, detect anomalous behavior, and enforce compliance with security policies.

From a security point of view, the Istio architectural components is represented in Figure 4.2.

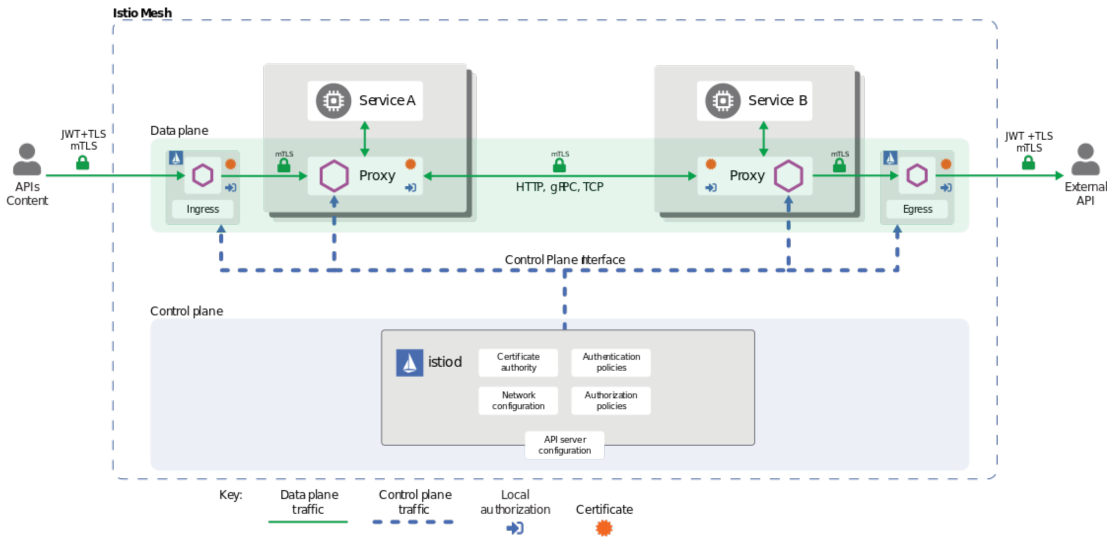


Figure 4.2: Istio Security Components (Source: Istio Documentation)

Figure 4.2 shows the main security component involved in Istio. It is clear how the control plane comes with a built-in Certificate Authority, that makes easy the key and certificate management of workloads. A different X.509 certificate is produced, signed and injected within each Envoy instance, and will be bound only to that specific workload identity. Each proxy acts as an intermediary between the microservice and incoming/outgoing traffic, implementing security measures and upgrading outgoing plaintext traffic to encrypted traffic, or decrypting incoming traffic before passing it to the application. This reflects the general policy-based access control model defined by NIST (Chapter 3, section 4): each sidecar and perimeter proxies (i.e. ingress and egress gateways) thus work as Policy Enforcement Points (PEPs) placed in front of each microservice and implement workload-specific authentication and authorization policies upon instructions sent by a control plane that acts instead as a PDP [31]. The Envoy proxy thus represents a transparent infrastructural layer that allows PEP to be shifted around each individual workload and shrink the implicit trust zone, reducing the attack surface of individual services and applying specific security policies to them.

This made such a solution even more interesting and suitable for our case. What is extremely important is that the sidecar model adopted by Istio allows each workload to maintain its security posture even when migrating from on-premises data centers to nonenterprise cloud instances: this is possible because the PEP is lifted and moved along with the application container within the pod, and thus the authentication and authorization policies are still enforced regardless of the infrastructure in which the services reside.

4.1.3 Identity

As seen in Chapter 4, identity is a fundamental concept of any zero-trust architecture. Continuous verification and validation of the identities of workloads and users interacting with a given service are essential to eliminate implicit trust and ensure that only those who are actually authorized can communicate with a given workload, even if they are internal to the enterprise.

Public key certificates are a key component of modern security infrastructure, providing strong

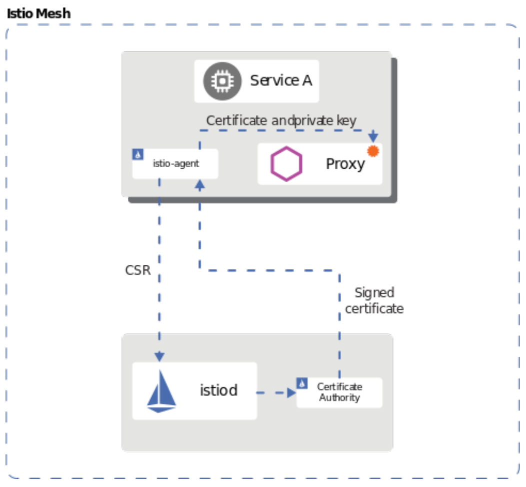


Figure 4.3: Istio Certificate Management Schema (Source: Istio Documentation)

assurances of identity in many contexts. When properly implemented and managed, they offer robust mechanisms for verifying the identity of entities such as websites, servers, and individuals in online transactions.

Istio employs a certificate authority (CA) built into the control plane to issue X.509 certificates for service identities. Istio's CA is responsible for generating, signing and distributing certificates to services within the mesh. Istio automatically injects sidecar proxies with these certificates. In fact, in the same container where the envoy resides, it also runs an istio agent that cooperates with istiod to automate key and certificate rotation: it is booted before Envoy, and is responsible for creating and managing the public-private key pair, and requesting a valid certificate from the control plane via certificate signing requests (CSR). If the CA validates the CSR, a valid certificate is produced and sent to that agent. Then, when the proxy starts, it will retrieve through the agent the key and the corresponding certificate to be used, thus having the assurance that when it starts envoy always has a valid certificate ready to use.

The Istio agent continuously monitors workload certificate expiration, and this process is repeated periodically to ensure certificate and key rotation, fully automating identity management of the various microservices in the service mesh. These certificates are then used for mTLS authentication between services, and consequently implementing identity-based authorization policies. [31]

In Kubernetes, the Istio identities embedded in the certificates are automatically associated with the Kubernetes service accounts mounted by the pods running the workloads, allowing services to be uniquely identified regardless of their network location or IP address. This enables secure communication between services regardless of their deployment environment, meeting the security requirements of a zero-trust network and enabling their interoperability across clusters and clouds. Consequently, to ensure that each Deployment kubernetes has its own distinct identity, cluster administrators should take care to create different Service Accounts for each microservice. Obviously, in the case of ReplicaSet, each replica will have the same identity/service account.

4.1.4 Service Mesh Observability

Istio provides several extra components and features for providing observability into the service mesh, relying on the logs and metrics collected and sent by each Envoy proxy each time it intercepts traffic. These components include:

- *Prometheus*: Istio integrates with Prometheus, an open-source toolkit for monitoring and alerting, to collect and store metrics about the service network and the applications running within it. Prometheus collects metrics from Istio components such as Envoy proxies. These metrics include traffic metrics (such as request speed, latencies, and error rates), resource utilization metrics, and more.

- *Grafana*: Grafana is a popular open-source visualization and analysis platform that works seamlessly with Prometheus. Istio includes preconfigured Grafana dashboards that display metrics collected by Prometheus, providing insights into the performance and behavior of the service mesh and its applications. These dashboards provide visibility into traffic patterns, latency distributions, error rates, and other important metrics.

- *Jaeger*: Jaeger is an open-source distributed tracing system that provides information about the flow of requests from microservices. Istio integrates with Jaeger to collect distributed traces from Envoy proxies and correlate them across multiple services. Jaeger makes it possible to track requests as they propagate through the service network, identify latency bottlenecks, diagnose errors, and understand the end-to-end flow of requests.

- *Kiali*: Istio includes an integrated Web-based dashboard called Kiali that provides a high-level overview of the service mesh and its components. The dashboard displays information on traffic routing rules, security policies, and telemetry configurations applied to the service mesh. It also includes views of traffic flow between services, traffic distribution between versions of a service, and other relevant information.

4.2 Palo Alto CN-Series Containerized Firewall

As already mentioned in Chapter 3, implementing a Zero-Trust network does not involve abandoning the castle-and-moat defense approach, since both physical and virtual FWs play an indispensable role in securing on-premises and cloud deployments. On the contrary, it means implementing the principle of defense-in-depth by integrating firewalls and IDS/IPS with other types of countermeasures. Unfortunately, however, the Kubernetes network model raises new challenges concerning the north-south traffic entering or leaving a pod within the cluster. Whenever a pod send traffic to destinations external to the cluster, because of the network address translation (NAT) performed by the host (typically using its own iptables) on the packets, all outgoing traffic carries the node IP address as the source: thus, the IP address of the actual source, i.e. the specific pod, is unavailable. As a result, firewalls sitting outside the Kubernetes clusters are blind to the actual source of the traffic. At the same time, for effective security in a containerized environment it is essential to know the true source address before NAT, but the only way of doing this is by moving the firewall (or in general any security node) inside the Kubernetes cluster for maximum effectiveness.

That was the idea embraced by the CN-Series firewall, a next-generation firewall specifically designed by the Palo Alto Networks to meet the company's security requirements and protect their Kubernetes environments from modern threats, application attacks and data exfiltration. It consists of a containerized firewall which can be deployed within the cluster as a pod, intercepting and authorizing the incoming traffic just before entering the pod, as well the outgoing traffic as it leaves the pod, also ensuring source visibility before being natted or tunneled by the node. Unlike a service mesh, therefore, it represents a specific security solution that can be integrated within a Kubernetes cluster to overcome the security challenges posed by such a dynamic containerized environment, achieving visibility and security for containerized application workloads on Kubernetes clusters.

Unlike firewalls that operate at the L3 and L4 layer of the OSI model, which inspect and filter traffic using IP addresses, protocols and ports and perform stateful inspection, the Palo Alto Networks Container Native Firewall is actually a Next Generation Firewall (NGFW). This means that it operates at the application layer of the OSI model (L7), inspecting the entire contents of each packet, including application data, URLs and headers, thus providing the highest level of visibility and control over network traffic. In addition to this capability, the CN-NGFW is able to offer several security features. Advanced URL filtering, combined with DNS security, helps prevent outbound connections to potentially malicious websites, including repositories containing malicious code. This, combined with full visibility into the packet payload, also helps prevent the exfiltration of sensitive data. Deep Packet Inspection (DPI) performed by CN-NGFW also enables Advanced Threat Prevention functionality, protecting inbound, outbound and east-west traffic, providing comprehensive protection against exploits, malware and command-and-control. In addition, the traffic content inspection performed by CN-NGFW can also impact TLS-encrypted traffic, allowing the firewall to decrypt (and possibly re-encrypt) packets and ensure that those containing malicious payloads are immediately identified and blocked.

Being able to identify and block specific applications, services or protocols, the CN-Series firewall is then able to implement and enforce more granular security policies based on application behaviour and content: this implies that true workload-to-workload micro-segmentation can be achieved, allowing only the traffic of a few specific services between two pods/workloads, while blocking everything else. This, together with Advanced Threat Prevention, allows communications between containers to be strictly protected, preventing threats that infect a particular workload from spreading laterally to other Kubernetes workloads or the rest of the infrastructure.

The aim of the CN-NGFW is, therefore, to provide full visibility and protection to all incoming, outgoing and east-west traffic of a cluster, but the most relevant consideration is that since it can be deployed as a pod within the Kubernetes environment, the security capabilities of the CN series are guaranteed regardless of the underlying infrastructure: this type of protection is always achieved, regardless of whether the containerized workloads are hosted on an on-premise data center or on a public cloud platform, or through a hybrid model. The CN firewall can therefore act as a PEP to shrink the implicit trust zone around each individual workload/pod, continuously inspecting incoming and outgoing traffic, having complete visibility over them and applying appropriate authorization policies based on their content and properties. It also does not slow down the deployment process, always ensuring the security posture of the pods behind him, even when they are destroyed and recreated, or scaled horizontally. In addition, it overcomes the limitation of traditional network security solutions on Kubernetes traffic visibility and makes it easy to deploy firewall solutions even on cloud infrastructures. [32][33]

4.2.1 CN-Series Core Building Blocks And Architecture

The CN-Series firewalls consist of several components, which interoperate in order to provide the functionalities mentioned before:

- Panorama
- Kubernetes Plugin
- CN-NGFW
- CN-MGMT
- PAN-CNI

In short, the architecture can be summarised in two main logical components: a security control plane and a data plane.

Panorama is the component that acts as the control plane or PDP for the entire security system, allowing cluster administrators to correctly configure network traffic policies using a single centralised security policy management system, and that can be deployed either in on-premise environments or in the public cloud as a virtual machine or on a physical server within the network. It is responsible for licensing the containerised firewalls deployed within the cluster and sending them configuration and security policies: to do this, it is securely authenticated and connected (using certificates) to the firewall management plan pods (CN-MGMT) deployed within the cluster, which then configure the actual firewall pods (CN-NGFW) on behalf of Panorama's instructions. [34]

Panorama must be integrated and work with a specifically designed **Kubernetes plugin** to allow it to gain visibility into container activity within a cluster. The K8s plugin is provided with service account credentials that it leverages to communicate with the API server and retrieve metadata about K8s resources in real-time. In this way, through the plugin, Panorama continuously monitors the cluster and the K8s objects within it, always knowing when a K8s resource or pod is created or destroyed.

Therefore, the Kubernetes plugin automatically collects information on existing namespaces, services, deployments, ReplicaSets and associated identifying attributes defined in each YAML file used to deploy K8s resources (such as the namespace in which they reside, or the application ports exposed by each pod or service, and the K8s labels assigned to each resource). In this way, Panorama can be constantly aware of the pods and K8s resources distributed within the cluster, as well as their IP addresses and namespaces. Using the collected information, the plugin automatically creates logical labels for each object (and corresponding pods) in the cluster, keeping track of an up-to-date correspondence between Kubernetes logical resources and which IP addresses are used by the pods representing each resource. Subsequently, these labels can be used to define security policies to authorise traffic entering or leaving the pods: this IP-address-to-label mapping thus simplifies the management of authorisation policies, as for each policy, it is sufficient to specify the source and/or destination label, and Panorama will automatically translate these labels into one or more policies affecting the various associated pods, and will continuously keep track of changes within the cluster (e.g. whenever pods are restarted or new ones are created at deployment time or for scaling reasons) and properly modify the authorisation policies using the IP addresses used by the new pods. [34]

The CN-MGMT and CN-NGFW represent the management and data plan of the containerised firewall, respectively: they have been separated to improve runtime protection workloads and to support a smaller footprint. The **CN-MGMT** is deployed as a pod in the cluster and is responsible for managing the NGFWs and, thus, licensing and configuring security policies on each firewall on behalf of Panorama. Actually, it is deployed as a StatefulSet ensuring volume persistence and configuration synchronization, and it is exposed as a K8s service. Typically, at least two CN-MGMT pods are deployed in the cluster to ensure fault tolerance, as a single CN-MGMT is capable of managing existing CN-NGFW pods in the event of a restart or failure of a CN-MGMT pod. Each NGFW is connected to one CN-MGMT pod using an IPsec tunnel,

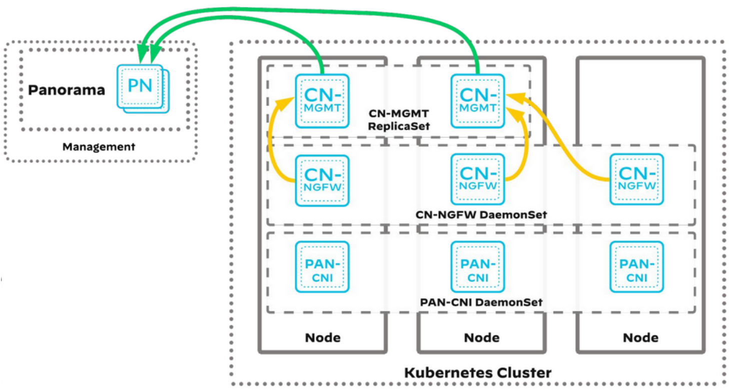


Figure 4.4: An Overview of CN-series components in a DaemonSet Mode Deployment

granting secure communication. The **CN-NGFW** pods, on the other hand, are the actual PEPs that intercept and inspect pod traffic and then enforce the authorisation policies received from the PDP by exploiting the IP address and tag mapping created for the pods, nodes, namespaces and services deployed in Kubernetes.

Finally, the **PAN-CNI** is what makes it all work: it is a CNI plugin specifically designed to be inserted into the CNI chain of each node within the cluster, with the objective of directly managing and customising the process of allocating pod network interfaces. In fact, to put the PEP in front of each pod and ensure its security posture, it is sufficient to tag the corresponding Kubernetes resource (using its YAML file) or the namespace in which it is deployed with a specific label: this will result in the CNI plugin reading the annotation on each application pod as it arrives to determine whether or not to enable security, and possibly configure its network interface to redirect traffic to the CN-NGFW pod for inspection as it enters and exits the pod. This ensures that traffic cannot bypass the PEP, since the security node is decoupled from the application pods and, and due to the proper network configuration provided by the PAN-CNI, traffic must necessarily pass through the firewall and be explicitly authorised before entering or leaving the 'trusted zone'. [34]

4.2.2 Deployment Modes

The firewall can be deployed within the cluster using two main modes: DaemonSet or Kubernetes Service, depending on the enterprise's performance requirements.

When deployed in **DaemonSet mode**, a different CN-NGFW pod instance is deployed on each node of the cluster where the application pods reside. In this case, as can be seen from Figure 4.5, the CNI configures the network interface of each newly created application pod to redirect traffic to the firewall using virtual wires. Each firewall pod has 60 vwire interfaces, and each application pod to be protected requires a pair of these interfaces, which are connected together internally to the FW, acting as a sort of physical network wire for the traffic. Since the vwire interfaces do not have Layer 2 or Layer 3 network addresses, the firewall does not participate in routing or switching the infrastructure, and therefore, if security policies allow traffic, packets will transparently pass through this wire: each time a packet enters a vwire interface, it enters the firewall pod and, once permission is obtained, it can only 'follow the wire', necessarily exiting the other corresponding one linked to the first. Each pair of vwire interfaces comprises a "trust"

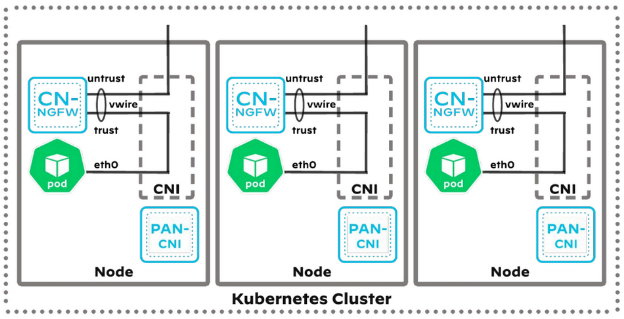


Figure 4.5: DaemonSet Mode Deployment

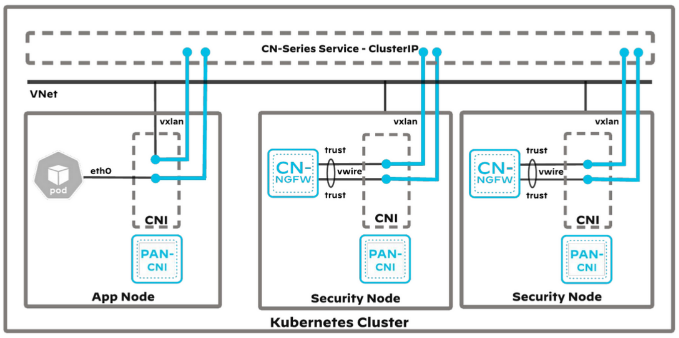


Figure 4.6: Service Mode Deployment

interface and an "untrust" interface, where the "trust" interface is connected directly to the pod (the trust zone), while the "untrust" interface is connected to the Kubernetes overlay network (the untrusted zone). In this way, the firewall acts as a PEP, reducing the trusted zone to a single pod and separating it clearly from the rest of the network. This mode, however, limits the scalability of the security nodes, as each CN-NGFW can connect and protect a maximum of 30 pods, so it should only be used when you have a cluster with a large number of nodes or few pods. On the other hand, it offers low latency and high firewall capacity. [34] [35]

On the other hand, the **Service Mode** allows the firewall to be deployed as a Kubernetes service: in this case, the firewall pods can be deployed on a separate dedicated node (the security node, as seen in Figure 4.6), and thanks to the properties of the Kubernetes service, they can automatically scale up while maintaining a stable virtual IP address (a clusterIP), and an appropriate number of NGFWs can be spin up and down according to current needs. The configuration adopted by the CNI is slightly different in this case: VxLAN tunnels are used to connect the network interfaces of the application pods that need to be protected to the CN-NGFW K8s service, redirecting incoming and outgoing traffic to one of the backend pods for inspection before entering or exiting the pod. [34] [35]

Finally, the **PAN-CNI** is what makes everything work: it is a CNI plugin specifically designed to be inserted on the CNI chain of each node within the cluster, with the aim of customizing the allocation process of pods network interfaces. In fact, in order to put the PEP in front of each pod and ensure its security posture, it is enough to label the corresponding Kubernetes resource (using its YAML file) or the namespace where it is deployed: this will involve the CNI plugin reads the annotation on each application pod as it comes up to determine whether to enable security or not, and eventually configure its network interface to redirect traffic to the CN-NGFW pod for inspection as it ingresses and egresses the pod. This ensures that the traffic cannot bypass the PEP, since the security node is decoupled from the application pods, and due to the network configuration, traffic must necessarily traverse the firewall and be explicitly authorized before entering or leaving the "trusted zone".

Chapter 5

Testing Environment Design, Installation And Configuration

After studying the basic concepts and state-of-the-art network security solutions, we needed a test environment that would allow us to recreate a realistic scenario that we could exploit to achieve our goal. The first step was to install a Kubernetes cluster, and thus set up a set of machines on which to install our orchestrator and deploy and run microservices-based applications. I then proceeded to design and develop a small web application that adopts concepts and technologies typical of microservice architectures, with a specific focus on the backend tier: this allowed me to get to know the application well and have full control over the various aspects, especially how these services interact with each other. After that, I designed and implemented a trivial IAM service, which is strictly necessary to authenticate and authorize the users of my application. Finally, I moved on to define and configure the Kubernetes resources needed for the orchestrator to deploy, run, and manage my applications on the cluster

5.1 Kubernetes Cluster Installation

Kubernetes deploys and manages applications (also called workloads) by placing them within pods and running these entities on the cluster nodes. The basis of everything, then is to have a cluster of nodes, a set of machines whose purpose is to provide the computational resources needed to run our services and the orchestrator that will manage them. Each cluster node can be a virtual or physical machine, depending on the cluster you have or want to create. Nowadays, we can find on-premise Kubernetes clusters, where organizations and datacenters have their own clusters of nodes on which to install the orchestrator and run workloads, or you can refer to cloud providers by asking and paying them for a set number of nodes and resources for each. In the specific case of this work, Spike Reply already had an on-premise server and testing environment with discrete compute capabilities and a VMWare hypervisor that easily allowed the underlying resources to be virtualized. The availability of proprietary resources without additional costs and the advantage of being able to have fine control and complete access over the resources used by the nodes and various configurations as needed (and thus being able to inspect and test various aspects at will), led us to create a cluster of virtual machines that would serve as cluster nodes. Typically, a cluster is composed of several nodes in order to have an amount of computational resources to ensure a good level of availability and resilience of the applications run, reducing any downtime. But considering the limited amount of resources to be virtualized and that the purpose of the cluster is solely to obtain a proof of concept of zero-trust networks, we opted for a fair and realistic compromise by going for a cluster of 3 nodes, one of which is dedicated to run the master node and two worker nodes where our pods will be run. Thus I moved on to create three VMs that make up our cluster, installing Ubuntu Server 20.04 OS on each of them and configuring SSH remote access via an asymmetric challenge-response authentication, using my own private key. Each node was assigned 12 GB of RAM and 4 vCPUs, a different IP address and a unique hostname to be uniquely identified within the cluster. Subsequently I moved on to install the Kubernetes platform and its components: each node in the cluster must necessarily have some basic components installed, such as kubelet, kube-proxy and a runtime container. For our test environment I adopted version 1.23 of Kubernetes, which is one of the most stable and broken-in versions.

There exist several tools you can use to deploy your own Kubernetes cluster, like:[36]:

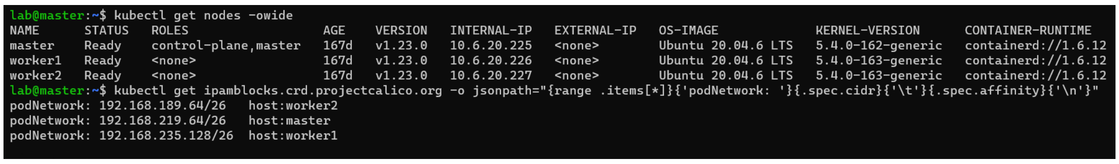
- **Minikube**, an open-source tool which permits setting up a local Kubernetes cluster made of a single node on your own laptop, and typically used for learning and testing purposes
- **Kubeadm**, which requires as a minimum a cluster of two nodes and requires a given expertise degree
- **Kind**, another tool to deploy a Kubernetes cluster locally inside a docker container
- **K3S**, a light Kubernetes version created for production use on small servers, IoT appliances, etc.

I chose to use **Kubeadm**, since allowed me to bootstrap and administrate a minimum on-premise Kubernetes cluster that conforms to best practices and gave me the possibility to discover the full potential of Kubernetes. Kubeadm is a tool whose aim is providing commands like **kubeadm init**, **kubeadm join** , **kubeadm upgrade** etc. as best-practice fast and easy way for installing Kubernetes clusters [37].

First of all it was needed to install the container runtime on each node, so that pods can be run on them: we opt for containerd, which has been designed to be lightweight and focuses on executing containers reliably and efficiently. I then moved on to installing the kubelet and kubeadm services on each node, and subsequently I installed kubectl on the master node. Kubectl is a command-line tool that allows you to execute any kind of command to Kubernetes clusters, permitting you to deploy applications, inspect and manage cluster resources, view logs and other stuffs. It represents the starting point for creating our testing environment and performing any kind of action on Kubernetes. [36]

In order to initialize the Kubernetes control plane node, I launched the kubeadm init command on master node. The penultimate step was to install the CNI that we want to use for implementing the Kubernetes Network Model: the choice was Calico, which represents a plug-in that implements the Kubernetes Container Network Interface (CNI) and, by leveraging some Kubernetes agents, provides networking features for containers and pods. Calico does not stand out for its simplicity but for its performance, reliability, and versatility. For these reasons is nowadays one of the most widely adopted and the popular CNI plugin. Calico set up a flat Layer-3 overlay network and assigns a fully routable IP address to every pod. It divides a large network CIDR (Classless Inter-Domain Routing) into smaller ranges of IP addresses and assigns one or more of these smaller blocks to nodes in the cluster. [38]

The default IPv4 address pool created by Calico at startup, unless otherwise specified, is 192.168.0.0/16. Each node is assigned a smaller subset of this pool, and the IPs of the Pods that



```
lab@master:~$ kubectl get nodes -owide
NAME      STATUS   ROLES                  AGE    VERSION   INTERNAL-IP    EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
master    Ready    control-plane,master   167d   v1.23.0   10.6.20.225    <none>        Ubuntu 20.04.6 LTS   5.4.0-162-generic   containerd://1.6.12
worker1   Ready    <none>                 167d   v1.23.0   10.6.20.226    <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.6.12
worker2   Ready    <none>                 167d   v1.23.0   10.6.20.227    <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.6.12
lab@master:~$ kubectl get ipamblocks.crd.projectcalico.org -o jsonpath="{range .items[*]}{'podNetwork: '}{.spec.cidr}{'\t'}{.spec.affinity}{'\n'}"
podNetwork: 192.168.189.64/26   host:worker2
podNetwork: 192.168.219.64/26   host:master
podNetwork: 192.168.235.128/26  host:worker1
```

Figure 5.1: Kubernetes cluster nodes

will be deployed on a specific node will be chosen from the range of addresses assigned to that node. We configured Calico in IP-in-IP mode, which means creating an IP-in-IP tunnel between each pair of nodes in the cluster, and thus between each subnet hosted on the node. Whenever a pod sends a packet that needs to leave the current node and reach another within the cluster, it is encapsulated with the IP-in-IP header and the node's IP address is used as the source. In this way, the infrastructure router does not see the IP addresses of the pods. [38]

Finally, once the Kubernetes network configuration was set up, I moved on to run the 'kubeadm join' command on the two worker nodes to make them part of the Kubernetes cluster, and thus nodes fully managed and monitored by the master node.

Once the installation is finished, the situation of the created cluster is shown in the Figure 5.1

5.2 Microservices-based Application Design and development

To make a realistic scenario suitable for implementing my Proof of Concept, I decided to develop a small web application following the microservice approach described in Chapter 2. It was realized web-based information system that supports a public transport company in managing the ticketing process to access to the transport vehicles. The system will support two kinds of human users, travelers and administrators. Travelers will be able to register and create an account by providing a valid e-mail address they are in control of. Once logged in, travelers can manage their profile, buy tickets and travelcards, consult the list of their purchases and download single travel documents in the form of QRCodes encoding a JWS (JSON Web Signature).

Administrators (i.e., employees of the transport company) will be enrolled via an administrative end-point by other administrators (provided they have the enrolling capability). At installation time, a single administrative username/password with enrolling capability will be created in order to bootstrap the process. Administrators can manage ticket and travel card types, by creating, updating and modifying their properties (validity period, price, usage conditions) as well as accessing traveler information and the related ticket purchased.

The application capabilities and features described above have to be implemented following the microservice-architecture approach. Therefore, the first step was to identify the application business domains and decompose the monolithic system into a series of smaller and loosely coupled services that are organized around business capabilities. Four different subdomains were identified, and consequently four distinct microservices were necessary:

- *IAM Service*
- *Traveler Service*
- *Catalogue Service*
- *Payment Service*

IAM Service is the one in charge of managing new user registration and their information, as well as credential recovery and system user authentication through a login endpoint. It will be discussed in more detail in Section 5.3.

Traveler Service is in charge of managing the identity and personal data of the travelers registered to the web portal, and also keeping track of the corresponding tickets purchased through the Catalogue Service. The service will be accessed via a REST API, which exposes various end-points accessible by authenticated users only, and provides only information pertaining to the identity of the client, some of which are accessible only to users having administrative privileges. The Traveler Service indirectly cooperates with the Login Service: the latter exposes a REST API consisting of some endpoints for registering a new user and validating their email address as well as a login endpoint, which accepts user credentials (username and password) and returns, when authentication is successful, a JWT temporarily representing the identity and role of the user. A registered user will have first to contact the Login Service, posting their credentials to the login endpoint and obtaining a JWT valid for the next hour, in order to be able to contact the TravelerService later, specifying the JWT as a value of the Authorization HTTP header. Upon reception of a new request, the TravelerService will validate the JWT (relying on a shared secret kept into the service properties, which should be the same used by the LoginService to emit the JWT) and, if successful, let the request be handled by the service business logic. Moreover, it cooperates with Catalogue Service when a ticket purchase request comes, and if the operation is successful, it generates and store within the Database the purchased tickets for that specific user.

The Traveler Service, as well as other microservices, have been implemented using the Kotlin language and the SpringBoot Framework, which simplify the process of building and deploying production-grade, enterprise-level applications. Traveler Service listens on port 8080, and exposes the following endpoints:

- *GET /my/profile*: this endpoint returns the current user's profile information (name, address, date of birth, telephone number, etc)
- *PUT /my/profile*: this endpoint allows the update of the current user's profile information.
- *GET /my/tickets/*: this endpoint returns a list of all the tickets purchased by the current user.
- *POST /my/tickets/*: this endpoint accepts a JSON payload containing information about the ticket type, transport zone, number of tickets to purchase and the validity period, and generates a corresponding number of tickets for the indicated transport zones. The generated tickets are stored in the service DB and will be returned as a payload of the response.
- *GET /admin/travelers*: this endpoint returns a list of usernames for which there exists any information (either in terms of a user profile or issued tickets). This endpoint is only available for users with the Admin role.
- *GET /admin/traveler/<userID>/profile*: this endpoint returns the profile corresponding to that userID. It is only available for users with the Admin role.
- *GET /admin/traveler/<userID>/tickets*: this endpoint returns the list of purchased tickets corresponding to that userID. It is only available for users with the Admin role.

Since each involves access to specific user information, all of the above endpoints are authenticated, which means that they mandatorily require requests to contain a valid JWT issued by the IAM service. The Traveler Service has its own Database, which consists of a Postgres instance listening on port 5432, containing 2 tables:

- The *Traveler Table*, containing user's information
- The *Tickets Table*, containing the user's tickets generated by the service at the time of purchase.

Payment Service is the one in charge of managing the payment operations during the ticket purchase process, handling payment data and interfacing with the banking service. It is actually a mock service, in the sense that it emulates a payment operation by simply "flipping a coin" and randomly deciding whether the payment transaction should be simulated as approved or not. It is contacted by the Catalogue Service whenever a user decides to purchase a ticket, to find out whether the purchase can be considered successful and tickets can be subsequently issued, or failed.

It listens on port 8080, and as we will see, since payment requests are managed through a message broker, it exposes only 2 endpoints (and both require a valid JWT, i.e. they are authenticated):

- *GET /transactions*: it returns all the transactions of the current user.

- *GET /admin/transactions*: It returns all the transactions of all users. It is only available for users with the Admin role.

The Payment Service has its own Database, which consists of a Postgres instance listening on port 5432, containing only a table named *Transactions* storing all the information about occurred transactions, both successful and failed.

Finally, the **Catalogue Service** is in charge of managing the sales process, providing the list of available tickets that can be purchased. It exposes various end-points, some accessible to everybody, some restricted to authenticated users and others accessible only to users having administrative privileges. This service cooperates with Traveler Service: when a purchase request is received, the Catalogue Service ask the Traveler one additional information about the customer user, which is then used to check compatibility constraints with sales available. If checks succeed, it further cooperates with both Traveler and Payment server: in case the acquisition process can be performed, the Catalogue Service saves the order information with status PENDING in its own database, and then sends back to the user a temporary status info. From this point, two things happen: it sends payment information to the Payment Service (the latter is in charge of sending payment information to the bank and establishing if the transaction is approved or not), and if the Payment Service provides back a "payment allowed" response, the Traveler Service is contacted in order to generate and store the tickets for that specific user, and the record with status PENDING is updated to ALLOWED (viceversa, with DENIED).

Catalogue Service listens on port 8080, and exposes the following endpoints:

- *GET /tickets*: this endpoint returns a list of all available tickets.
- *POST /shop/ticket-id*: this endpoint
- *GET /orders/*: this endpoint return a list of all user orders.
- *GET /orders/<order-id>*: this endpoint return a specific order. It can be used by the client to check the order status after a purchase.
- *POST /admin/tickets*: this endpoint can be used only by users with an Admin role to add catalog newly available tickets to purchase.

- *GET /admin/orders*: this endpoint can be used only by users with an Admin role to retrieve a list of all orders made by all users.
- *GET /admin/orders/<user-id>*: this endpoint can be used only by users with an Admin role to retrieve a list of all orders made by a specific user.

Except for the *GET /tickets* one, all the remaining above endpoints are authenticated, which means that they mandatorily require that requests contain a valid JWT issued by the IAM service. The Catalogue Service has its own Database, which consists of a Postgres instance listening on port 5432, containing 2 tables:

- *Tickets Table*, which contains records representing the various tickets and their information.
- *Orders Table*, through which manages purchase orders and related information.

Even if each service exposes its own REST API, the communication between Traveler, Payment and Catalogue services is managed through a Message Broker.

A message broker is an intermediary software component which provides a scalable, reliable, and flexible infrastructure for facilitating communication and message exchange between different applications or systems. It acts as a middleman, receiving messages from one application and delivering them to another, decoupling the sender and receiver. In microservices architectures, message brokers play a crucial role in facilitating communication between microservices. They enable asynchronous communication patterns such as event-driven architecture, where services publish messages and subscribe to them, allowing systems to communicate without the need for both parties to be available at the same time. This asynchronous nature promotes the scalability and flexibility of distributed systems, as well as their fault tolerance. Since this communication pattern is pretty common in real-case scenarios, it was implemented in the architecture to recreate a realistic context.

Apache Kafka was the message broker adopted. It allows several message queues to be implemented, and communication between services takes place through them: each service can act as a publisher or consumer. When it wants to contact other services, it acts as publisher and pushes the message inside the queue; the broker is then responsible for delivering the message to the services that have subscribed to that queue and want to consume the message inside it.

Because Kafka can scale horizontally by adding more brokers and partitions to distribute the workload, it relies strictly on a service called Zookeeper, which is a distributed coordination service used to maintain configuration information and provide distributed synchronization to the various possible instances of Kafka.

However, an API Gateway will be further placed in front of the various microservices using the Istio Ingress Controller, providing a unique endpoint that is able to manage the requests directed towards the various services REST API.

The trivial microservice architecture implemented for testing the security solution can thus be pictured as in Ffigure 5.2:

5.3 IAM Service Design and Development

A digital ticketing system needs to manage a set of users, securely storing their credentials and providing a way to recover them in case they are lost. This, in turn, entails that users have the ability to request a new registration, by posting a minimal set of personal information (chosen username, password, email address) to a */user/register* endpoint: as a consequence, some

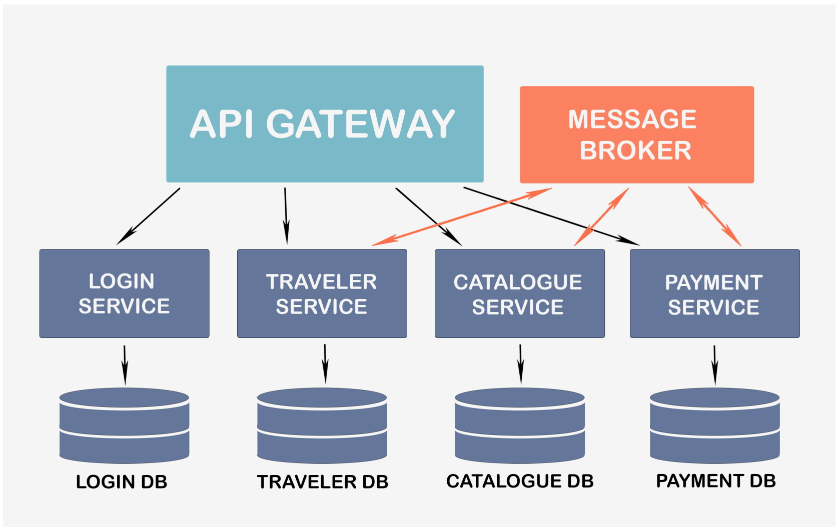


Figure 5.2: The microservice-architecture that implements the transport ticketing system

information will be stored and a validation process that will either lead to accept the supplied information, thus completing the registration, or to remove the request, dropping any supplied data, will be started. Since the registration process is open to the general public, care has to be taken to newly registered users. In order to prevent inconsistent data from being submitted, several checks will be enforced: username, password, and email address cannot be empty, email address must be valid, username and email address must be unique system-wide, and password must be reasonably strong. If any check fails, the request will be rejected with status code *400-Bad Request*; otherwise an entry will be stored inside a specific user DB table, containing the uploaded data, together with the indication that the record is not yet active, since there is no proof, yet, that the applicant has control of the submitted email address. A provisional random ID will be returned with status code *202-Accepted*.

In order to validate the given email address, when data have been recorded, a random activation code will be generated and stored in another db table, together with the provisional random ID, an expected activation deadline, an attempt counter, and a reference to the user record; moreover an email message will be sent to the provided address containing the activation code. If, within the expected time, a registration completion request is posted to the */user/validate* endpoint containing the provisional random ID and the corresponding activation code, the user record will be transitioned to the active state, the corresponding record in the activation table will be removed, and status code *201-Created*, will be returned, together with the newly created user information. If the request is received after the expiration of the deadline, the activation record will be removed from the activation table, and status code *404* will be returned. If the provisional random ID does not exist, status code *404* is returned. If the provisional random ID exists but the activation code does not match the expected one, status code *404* will be returned and the attempt counter will be decremented: if it reaches 0, the activation record will be removed together with the User entry.

Whenever a user wants to use any of the other authenticated microservices endpoints, they must necessarily log in first, providing their credentials (username and password) to a specific endpoint */user/login*, and it will return, when authentication is successful, a JWT temporarily representing the identity and role of the user.

JWT stands for JSON Web Token. It is a compact and self-contained way of transmitting information between two parties in the form of a JSON object. JWTs are commonly used for authentication and authorization in web applications, APIs, and microservices architectures. The payload of a JWT contains the claims, which are statements about an entity (typically the user) and additional data. These claims can include information such as the user's identity, roles, permissions, and any other relevant data.

In our case, the JWT will encapsulate the user claim and its role within the web system (i.e. simple user or administrator). Once the user performs authentication, the returned JWT can then be provided to other microservices to authenticate itself and access the API: this is done by enclosing the token within an HTTP header of the request, before sending it to the service. In our case each JWT token has a validity period, and expires after 1 hour from its issue.

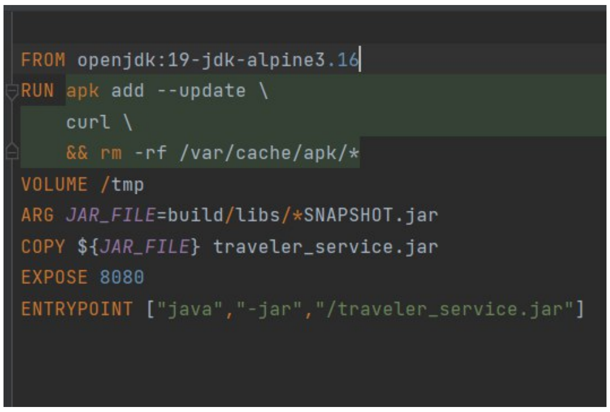
The loginservice has been provided with a public/private key pair: the private key is used to digitally sign the JWT token at its issuing, providing integrity and authenticity to the transmitted data, while the corresponding public key is used by microservices to verify and validate the JWT authenticity provided by the user. In this way, microservices can verify whether the provided identity is valid, by checking if the JWT has been tampered or it is expired. Typically a front-end is in charge of managing login operation and subsequently automatically placed within a given HTPP request to other microservices the JWT returned after the login: in our case, a front-end was not implemented, thus tests in the next chapters have been performed manually inserting JWT token in each HTTP request.

As with the other microservices, the SpringBoot framework and the Kotlin language were adopted for the implementation of this service, which for simplicity I called **loginservice**. The service listens on port 8081, and the REST API implemented and exposed by the service can therefore be listed as follow:

- *[POST] /user/register*: this endpoint is used to sign up into the web application. It accepts an HTTP POST request containing in the body nickname, email and password that the user wants to use. Depending on whether the various checks succeed or at least one fails, it returns a "202-Accepted" response with a provisional ID to be used for validation to follow, or "404-Bad Request"
- *[POST] /user/validate*: this endpoint is used to validate newly created accounts after registration. It accepts an HTTP POST request containing in the body the provisional ID returned from the /user/register endpoint, and the activation code sent via email. If the validation succeed, the account is activated and a "201-Created" response is sent.
- *[POST] /user/login*: this endpoint is used to sign in users into the application. It accepts an HTTP POST request containing in the body nickname and password of the user. Upon successful login, an HTTP response containing a valid JWT Token corresponding to the authenticated user is sent. Otherwise a "404-Bad Request" is sent.
- *[PUT] /admin/users/<userID>*: this endpoint only allows system administrators to assign new roles (like the "admin" role) to other simple users. It accepts an HTTP PUT request containing in the body the role to be assigned.
- *[GET] /security/iam/jwks*: this API exposes a JSON Web Key Set (JWKS). It provides a set of public keys that have to be used to verify any JSON Web Token (JWT) issued by this authorization server. This is also extremely useful to implement Istio request authentication in our PoC.

The IAM service makes use of a Postgres Database instance, with 3 tables:

- The *User Table*, containing information about the users (ID, date of birth, email, etc.)



```
FROM openjdk:19-jdk-alpine3.16
RUN apk add --update \
    curl \
    && rm -rf /var/cache/apk/*
VOLUME /tmp
ARG JAR_FILE=build/libs/*SNAPSHOT.jar
COPY ${JAR_FILE} traveler_service.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/traveler_service.jar"]
```

Figure 5.3: Dockerfile used for Traveler Service container image

- The *Activations Table*, containing temporary information like provisional random ID, attempts number, expiration, etc. about newly registered users that have not yet activated their accounts after registration
- The *Role Table*, containing information about roles provided to each user (it is extremely important during login to generate a JWT encapsulating user's roles and subsequently perform RBAC)

5.4 Kubernetes Resources Design and Configuration

Once we have installed our cluster and developed our microservice-based application and our authentication and authorization service, we need to define and create the Kubernetes resources needed to run our services within pods on the worker nodes of the cluster and make them accessible to each other. To deploy and run microservices inside Kubernetes pods, I needed first of all to create a Docker container image for each service I implemented and then upload them onto my private container registry on DockerHub from which they will then be downloaded upon deployment. A container image represents a binary file that encapsulates the application source code and all its libraries and software dependencies. [39]

Thus, I first required a Dockerfile to be written for each Springboot microservice. Although I could have used a more stripped-down base container image for security and performance reasons, each Docker file starts from an OpenJDK Alpine-based image (as I then chose also to install the curl tool on top of it to facilitate testing) and then copy the jar of the related service inside the image and use it as the entry point which bootstrap the microservice business logic. The fFigure 5.3 here is the Dockerfile used for creating the Traveler Service container image, although all the others are pretty much the same but they copy and use a different jar file:

For the Kafka, Zookeper, and Postgres services, however, I used the docker images available on the corresponding official DockerHub image registries.

The second step was to think about how to organize our microservices and Kubernetes resources within the cluster and then how many Kubernetes namespaces would need to be created for our applications. Namespaces are a way to divide cluster resources between multiple users. When installing a Kubernetes cluster, four namespaces are created by default: one namespace called **kube-system** (where all Kubernetes management system resources, such as the API server, scheduler and kube-controller, are distributed) and others called **kube-node-lease**, **kube-public** and **default**.

If you do not specify any namespace metadata for Kubernetes resources, they will be automatically assigned and placed within the **default** namespace, but as a best practice for a production cluster it is strongly recommended to follow the separation of concerns principle and to organize each different microservices-based application resources deployed on the cluster in a different and newly created namespace, adopting different labels for the various resources in order to distinguish them within the same namespace. [40]

Reason why I decided to create 3 new different namespaces by launching the ***kubectl create namespace <name>*** command from master node: *ticketapp, kafka and iam.*

The ticketapp namespace will be used to deploy and group the ticketing application resources, that is the microservices and their own databases Deployment, Services and ServiceAccounts, while IAM and kafka namespaces will be used to deploy resources related to the IAM application and resources needed to run the Kafka message broker and Zookeeper, respectively. As we will see in the next chapters, I will then go on to create an additional namespace for testing reasons. Note that in this case we are deploying the IAM service within the same cluster where applications will run, but in a typical production scenario it could be provided by a third-party provider and in general be deployed outside the cluster.

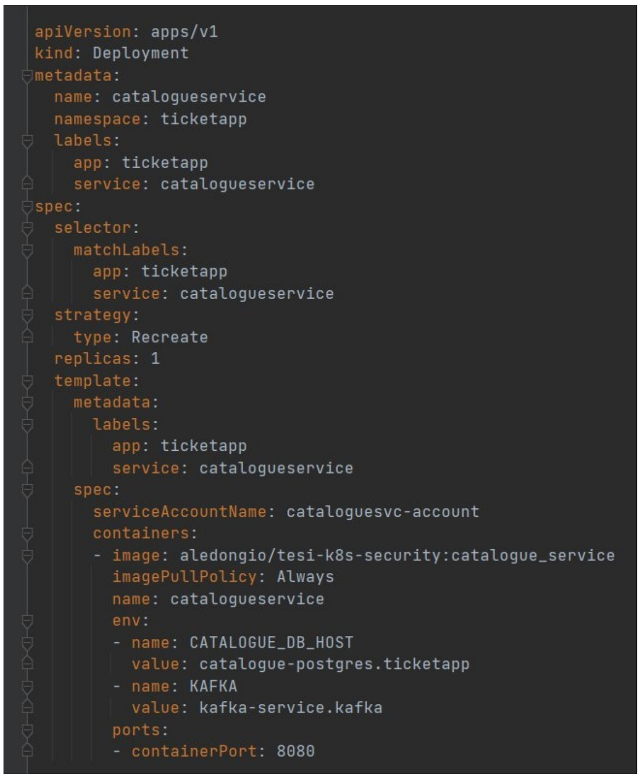
After all the container images were produced and pushed onto the DockerHub registry, it was necessary to write the YAML files needed to deploy and expose the various services on the cluster and make them communicate and function properly. For each service to be deployed, including kafka, zookeeper and postgres services, I have defined three different Kubernetes resources: one Deployment, one Service and a different ServiceAccount for each of them. Note that although I could have used a single ServiceAccount for each microservice-database pair, since they are tightly coupled, I preferred to use different SAs for each microservice and each postgres instance, because this allows for separate concerns and fine-grained authorization aspects for each service. In addition, as we will see in later chapters, ServiceAccounts will be extremely useful for providing some sort of identity and authentication to workloads running on the cluster.

Here (Figures 5.5, 5.6, 5.4) it is the three resources defined for the Catalogue microservice, but they are more or less the same for all the remaining services, with some slight changes:

Thus I created one single pod replica for each microservice, database instance and message broker service, even if in a production scenario typically we have multiple instances of each service for resilience and availability purposes. Then, since pods are ephemeral resources which are continuously appearing and disappearing and changing the associated IP address, I could not make each application call each other by simply using the Ephemeral IP of pods. Therefore I created a ClusterIP Service abstraction, which allows exposing our applications as network services within a cluster using unique stable IP addresses.

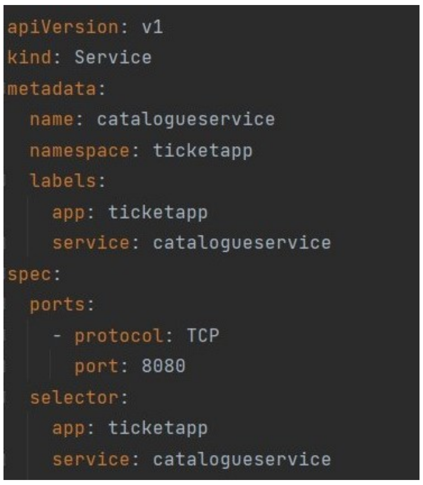
When Kubernetes creates a **ClusterIP Service** it assigns a virtual IP address which can be exclusively accessed from within the cluster, and therefore solely by the other Services running inside the cluster. This address is tied to the lifespan of the Service, and will not change while the Service is alive. Moreover, a DNS name will make corresponding to each ClusterIP service, adopting the **<ServiceName>.<Namespace>** notation. [41]

By using labels attached to each Deployment, I defined which pods and port will be backed to each Service, and then how is evinced from the Service Resource image here above, I assigned to each pod (i.e. to the corresponding Deployment resource) some environment variables containing the DNS name of each ClusterIP Service that will be used by pods in their binary to contact the postgres instances, the message broker and the zookeeper service.



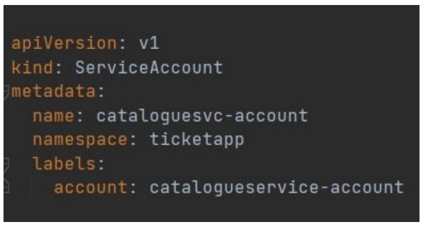
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalogueservice
  namespace: ticketapp
  labels:
    app: ticketapp
    service: catalogueservice
spec:
  selector:
    matchLabels:
      app: ticketapp
      service: catalogueservice
  strategy:
    type: Recreate
  replicas: 1
  template:
    metadata:
      labels:
        app: ticketapp
        service: catalogueservice
    spec:
      serviceAccountName: cataloguesvc-account
      containers:
      - image: aledongio/tesi-k8s-security:catalogue_service
        imagePullPolicy: Always
        name: catalogueservice
        env:
        - name: CATALOGUE_DB_HOST
          value: catalogue-postgres.ticketapp
        - name: KAFKA
          value: kafka-service.kafka
        ports:
        - containerPort: 8080
```

Figure 5.4: Deployment resource for Catalogue Service



```yaml
apiVersion: v1
kind: Service
metadata:
  name: catalogueservice
  namespace: ticketapp
  labels:
    app: ticketapp
    service: catalogueservice
spec:
  ports:
    - protocol: TCP
      port: 8080
  selector:
    app: ticketapp
    service: catalogueservice
```

Figure 5.5: Service resource for Catalogue Service



```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cataloguesvc-account
  namespace: ticketapp
  labels:
    account: catalogueservice-account
```

Figure 5.6: ServiceAccount resource for Catalogue Service

Currently, services are not exposed outside the cluster, but are only available to services running inside it. In order to do so, an Ingress Controller will be deployed in the next chapter.

5.5 Istio And Ingress Gateway Deployment

After deploying our Kubernetes cluster and configuring the Kubernetes resources for the various microservices, the next step was to install Istio. Once we download the Istio package, it brings with it an executable called *istioctl* that automates and facilitates the installation of the Service Mesh. By running the command **istioctl install –set profile=demo -y** it was possible to install the service mesh on the cluster in a jiffy, using the *demo* configuration profile. Istio offers several built-in configuration profiles (which are preferred for production environments). However, the demo was selected to have a good set of default features to test. Using such a configuration, Istio permits you to easily and automatically deploy a number of already configured (or partially configured) services, such as Prometheus, Grafana, Kiali and Jaeger instances. It also provides us a couple of pre-configured gateways named istio-ingressgateway and istio-egressgateway, that we can re-configure and use for managing, controlling and observing ingress and egress traffic.

They are all deployed within the *istio-system* namespace, a namespace created on purpose when installing Istio to deploy all the related resources, including the Istio Control Plane pods/services.

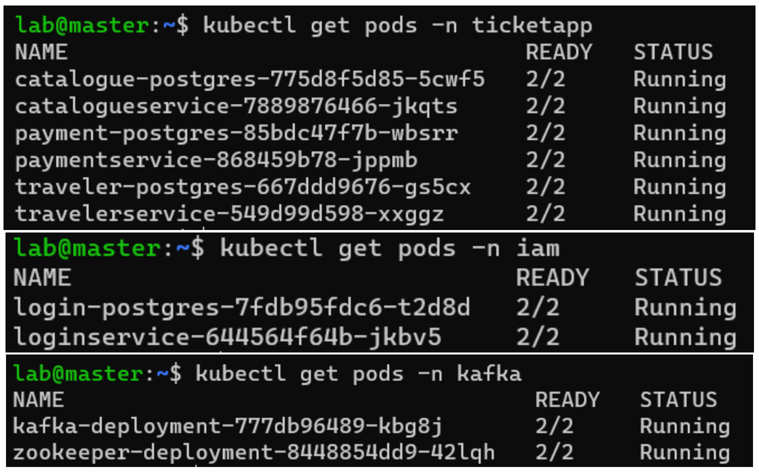
In particular, the built-in ingress gateway will allow us to expose HTTP services out of our cluster through an endpoint, and this will allow us to easily handle different types of tests, and most importantly, to recreate a full-fledged realistic scenario. We will only need to provide an appropriate configuration to route traffic to the various services in the cluster appropriately, and then it will map a path to a route at the edge of your mesh.

In order to be accessed from outside, the Ingress Gateway requires a stable IP address and port that can be easily reached. Since we did not have a third-party LoadBalancer, we opted to use a Service NodePort to expose our reverse proxy: thus, a host port was selected that each host should exclusively reserve for the ingres gateway service, automatically remapping and redirecting incoming traffic on that host port, to the pod running the ingress gateway. Istio's Ingress gateway can act as a TLS gateway and also perform authentication and authorization of incoming HTTP and TCP requests, as Envoy's proxies for each pod in the cluster also do. Next I also exposed the Kiali, Prometheus and Grafana services as NodePort so that it can be easily accessed as a web service from outside. There is no need to manually inject Envoy proxies within each pod. In order to allow Istio to inject Envoy sidecar proxies when deploying the application automatically, simply label the namespace to which the application will belong with *istio-injection=enabled*: for example, for the ticketapp namespace we have to run the command **kubectl label namespace ticketapp istio-injection=enabled**.

I then labeled the various namespaces ticketapp, kafka, iam, and default, and then deployed the Kubernetes resources associated with the various microservices in the namespaces so that when they started, a proxy envoy would be automatically injected into each pod. Figure 5.7 shows the states of the pods running in the cluster immediately after their deployment, in the ticketapp, kafka and iam namespace: note that each pod in these namespaces executes 2 containers, one represented by the main application container and the other represented by the injected sidecar. Note also that we have 2 pods associated with each microservice, representing the business application logic and the related database instance. Note also that the zookeeper service, which is strictly necessary for kafka to run, also runs in the kafka namespace. Each application and database pod mounts and uses a separate purpose-built service account: this ensures that the certificate injected into each service's proxy is tied to a unique identity that will allow fine-grained policies to be applied to microsegment traffic between services.

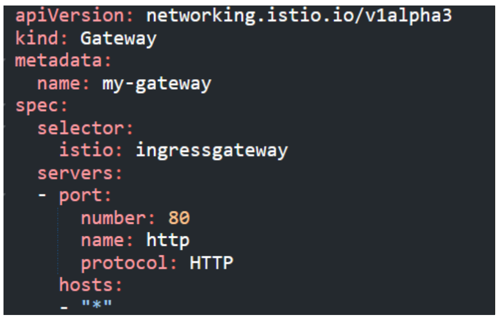
To enable the ingress gateway to properly route the received traffic to the various services in the mesh, Istio is required to configure two Kubernetes Resources: a *Gateway* and a *VirtualService*. The first describes a load balancer located at the edge of the network, which acts as an API gateway receives all incoming HTTP/TCP traffic and allows it to specify which connections can enter the network. The second, on the other hand, must contain the specific information that the gateway must use to forward incoming requests and reach the correct destination service within the cluster (Figure 5.8 and 5.9).

Figure 5.8 shows the Gateway resource provided: the gateway is now able to accept only HTTP traffic on port 80. Figure 5.9 instead, shows a part of the original VirtualService deployed, since it was too long: highlighted here is the configuration for handling traffic directed to the login and traveler service within the cluster. In this way, any HTTP traffic directed to services within the cluster can be sent to the IP address and port of the gateway, which serves as the only point of contact with the outside world, and based on the specific URL path contained, it will be forwarded to one service or another on the specified port. In this example, whenever the URL begins with "/iamservice/," this prefix is removed from the actual path and the request is forwarded to the Login Service in the iam namespace, on port 8081. Conversely, if the URL



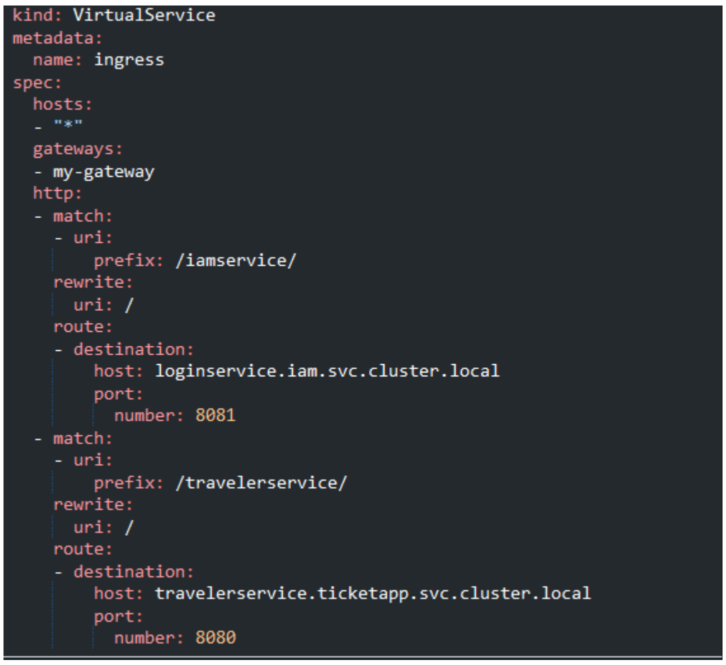
```
lab@master:~$ kubectl get pods -n ticketapp
NAME                                  READY   STATUS
catalogue-postgres-775d8f5d85-5cwf5   2/2     Running
catalogueservice-7889876466-jkqts     2/2     Running
payment-postgres-85bdc47f7b-wbsrr     2/2     Running
paymentservice-868459b78-jppmb        2/2     Running
traveler-postgres-667ddd9676-gs5cx    2/2     Running
travelerservice-549d99d598-xxggz      2/2     Running
lab@master:~$ kubectl get pods -n iam
NAME                               READY   STATUS
login-postgres-7fdb95fdc6-t2d8d   2/2     Running
loginservice-644564f64b-jkbv5     2/2     Running
lab@master:~$ kubectl get pods -n kafka
NAME                                    READY   STATUS
kafka-deployment-777db96489-kbg8j       2/2     Running
zookeeper-deployment-8448854dd9-42lqh   2/2     Running
```

Figure 5.7: Pods running in Ticketapp, Kafka and Iam namespace after Istio deployment



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

Figure 5.8: Gateway Resource



```
kind: VirtualService
metadata:
  name: ingress
spec:
  hosts:
  - "*"
  gateways:
  - my-gateway
  http:
  - match:
    - uri:
        prefix: /iamservice/
    rewrite:
      uri: /
    route:
    - destination:
        host: loginservice.iam.svc.cluster.local
        port:
          number: 8081
  - match:
    - uri:
        prefix: /travelerservice/
    rewrite:
      uri: /
    route:
    - destination:
        host: travelerservice.ticketapp.svc.cluster.local
        port:
          number: 8080
```

Figure 5.9: VirtualService Resource

starts with "/travelerservice/", the request is forwarded to the Traveler Service in the ticketapp namespace on port 8080. The same process also occurs for the other services.

Chapter 6

Zero Trust Network Model Implementation

In this chapter, we will attempt to implement the zero-trust principles explained in Chapter 4, using first Istio and then the containerized Palo Alto CN-Series firewall. Naturally, the proof of concept was implemented using the microservices of the ticketing application developed and deployed specifically. The results obtained and the validation of this implementation will be discussed in Chapter 7.

6.1 Istio

As far as Istio is concerned, its authentication and authorisation capabilities, provided by Envoy proxies after appropriate control plane configuration, were explored. In section 6.1.1 will seek to protect communications within the cluster and implement an authentication mechanism based on the identity contained in X.509 certificates and, consequently, workload-to-workload authorisation policies based on the identity extracted from them, in an attempt to micro-segment the network. In section 6.1.2, on the other hand, an attempt was made to implement a more fine-grained authorisation policy, focusing on the specific end-user and request instead of generic communication between two workloads.

6.1.1 mTLS, Peer Authentication And Authorization

One of the main features of Istio is the adoption of mTLS for protecting service-to-service communications. When an mTLS channel is established, it permits encrypting all the traffic sent over this channel, using a suitable ciphersuite. This guarantees the confidentiality of the communication. Moreover, for each message exchanged between the two parties, its authentication and integrity can always be verified (and in some sense, be guaranteed) through a Message Authentication Code (MAC) generated by the client. These two features provide protection against several Man-in-the-Middle (MitM) Attacks: It is not possible to eavesdrop on communications occurring over mTLS channels, let alone alter the messages exchanged without detection. In addition they also provide protection against replay and filtering attacks.

However, the most relevant security property in our case is Peer Authentication: in mutual

TLS it provides strong authentication. Peer Authentication occurs during the TLS handshake process (i.e., before the channel is opened): both the client and the server exchange certificates with each other to prove their identities. Before opening the communication channel, each party verifies the presented certificate to ensure its validity, that it was issued by a trusted CA (the Istio CA in this case), and that the identity information in the certificate matches the expected identity of the peer. This verification process helps ensure that the peer/workload is who it claims to be.

This means that both the client and the server authenticate each other's identities before proceeding with the communication. If the verification process of even one of the two certificates fails, the Authentication process is considered failed, and the channel will never be opened. Otherwise, the cryptographic keys to be used for the confidentiality and integrity of messages will be exchanged, and from here on all traffic exchanged through this channel will be implicitly protected, both parties being in the meantime sure of the identity of the other party.

Unless otherwise specified, Istio automatically configures each proxy to use mTLS when the corresponding workload contacts other services in the same Istio mesh (i.e., their Envoy proxies) and to use plaintext traffic when communicating with other workloads without Envoy proxies. This kind of auto-configuration is possible because Istio keeps track of the various services in the Mesh. This means that even if the outgoing traffic produced by the main container is in the clear, service-to-service communication is automatically upgraded to mTLS traffic by the client-side proxy. It implies that if authentication succeeds (i.e., everything concerning the certificate is ok), the client proxy is now aware of the identity of the server by extracting it from the valid certificate. Viceversa, the communication is immediately denied since it is not possible to trust the identity of the other party. The same thing applies on the server side, with the difference that in that case, the identity of the client is strictly necessary to verify and enforce authorization policies on the source: so all the more reason communication should be blocked since it is not possible to know who wants to access the resource.

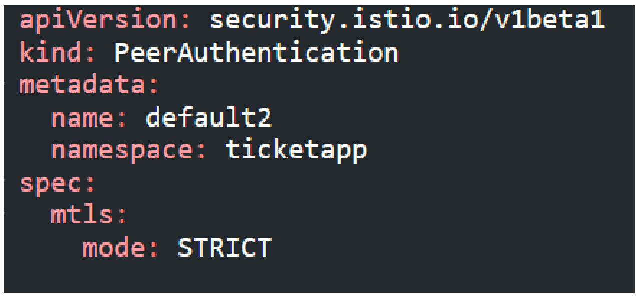
During the handshake, the client-side Envoy also does a secure naming check to verify that the service account presented in the server certificate is authorized to run the target service, providing stronger protection.

When acting as a server-side proxy, Envoy can be configured to operate as servers in different mTLS modes: PERMISSIVE, STRICT and DISABLE.

- *PERMISSIVE*: it configures the proxies to accept both plaintext and mTLS traffic
- *STRICT*: it configures the proxies to accept only mTLS traffic
- *DISABLE*: it configures the proxies to accept only plaintext traffic

By default, Istio automatically configure proxies in PERMISSIVE mode. However, even if the permissive mode is useful when operators want to migrate services to Istio without breaking existing plaintext communications, from a security point of view it should be avoided. With this mode, the proxy may allow plaintext traffic, and in such cases no authentication or authorization checks will be performed. Authentication is the first step towards microsegmentation: without it, we cannot know the identity of the source sending the request, and thus zero trust is never achieved. Thus is preferable to adopt only STRICT mode, which allows workloads to accept only connections from authenticated entities (thus applying the required security controls), and granting that the communication is protected from any man-in-the-middle.

In order to configure which mTLS mode a given workload has to adopt as a server, a *PeerAuthentication* has to be created and applied. It is a Custom Resource Definition introduced by Istio, which is used by the control plane to properly configure the authentication policy that the various proxies must adopt for incoming connections. Peer and request authentication policies use



```yaml
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default2
  namespace: ticketapp
spec:
  mtls:
    mode: STRICT
```

Figure 6.1: PeerAuthentication applied to ticketapp namespace

selector fields to specify the label of the workloads to which the policy applies. The authentication policy can be applied at different levels of granularity, each with a more or less specific target:

- *Mesh-wide*: it applies to all the workloads within the namespaces of the Service Mesh (i.e., properly labeled with "istio-injection=enabled")
- *Namespace-wide*: it applies to all the workloads within a specific namespace
- *Workload-wide*: it applies to a specific workload within a specific namespace, maybe on a specific port

It is possible to set a single mesh-wide policy and a single namespace-wide policy for each namespace. Multiple authentication policies can be set for a given workload at different layers, but Istio will always apply the narrowest matching policies for each workload. This means that if a DISABLE mode is applied to the whole namespace, and a STRICT mode is instead applied only to a specific workload X within that namespace, workload X will be configured in STRICT mode, while all the others with DISABLE. This provides better flexibility for policy configuration.

However, in our case, it is enough to apply either a Mesh-wide policy or a Namespace-wide policy to each namespace, in this way, all the workloads within them will only accept mTLS traffic. The second option was adopted.

Thus, a STRICT mTLS PeerAuthentication was applied to every namespace of my application (i.e., to the pods within them): ticketapp, kafka and iam. This implies that not only communication between microservices will be protected and authenticated, but also communications between microservices and kafka, as well as between microservices and their database instances, since they are TCP based ones. Figure 6.1 shows the PeerAuthentication resource set for the ticketapp namespace (for the other namespaces, they are pretty much the same, with only a different value of the selector field).

Attention must, however, be paid to the fact that Istio, unless a DestinationRule is specified, configures client-side proxies to use mTLS or not depending on the PeerAuthentication mode set for the various server-side workloads: this means that if the recipient is inside the Mesh and has an incorrect configuration in DISABLE mode, the outgoing traffic will not be authenticated and protected (i.e., plaintext). This behaviour can be overridden by setting a specific DestinationRule for each service in the mesh, as I did: figure 6.2 shows the DestinationRule specified for the Catalogue Service. This implies that Istio will also configure all proxies to mandatorily use mTLS with the specific target workload in the mesh (the catalogue service in the specific case of the figure) when acting as a client, resulting in a more secure configuration. The ISTIO MUTUAL mode specifies that proxies must necessarily use certificates generated automatically by Istio for mTLS authentication.

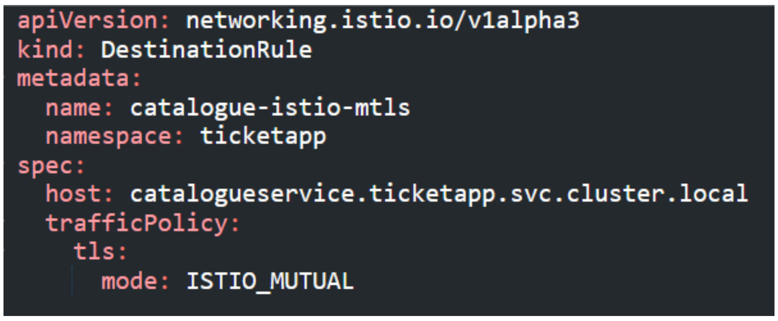

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: catalogue-istio-mtls
  namespace: ticketapp
spec:
  host: catalogueservice.ticketapp.svc.cluster.local
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

Figure 6.2: DestinationRule for the Catalogue Service

Once the mTLS connection has been successfully established, each party will then know the identity of the other extracted from the received and verified X.509 certificate, i.e. the Service Account of the workload. This retrieved workload identity can then be used by the server-side proxy to apply authorisation policies, and verify whether or not a given request from that specific source is actually legal, or should be rejected. Through the use of Istio's Authentication and workload-to-workload Authorization policy, I will therefore try to micro-segment communications within the cluster, ensuring that only services that actually need to communicate and cooperate are enabled to do so, while denying all other requests (both unauthorised and authenticated but unauthorised).

To micro-segment the traffic and ensure maximum security, a whitelisting approach was adopted: it means that in order to secure cluster connections and requests, a default-deny authorization model was adopted, which denies all requests by default and only accepts those that are explicitly allowed.

When there exists several Authorization Policies defined for the same workload at the same time, and these use CUSTOM, DENY and ALLOW actions, the CUSTOM action is evaluated first, then the DENY action, and finally the ALLOW action. The evaluation is determined by the following rules:

1. If there are any CUSTOM policies that match the request, evaluate and deny the request if the evaluation result is denied.

2. If there are any DENY policies that match the request, deny the request.

3. If there are no explicit ALLOW policies for the workload, implicitly allow the request.

4. If any of the ALLOW policies match the request, allow the request.

5. Otherwise implicitly deny the request.

However, as with an authentication policy, an authorization policy may have scopes with different levels of granularity to provide more flexibility in policy definition, such as mesh-wide, namespace-wide, and workload-wide policy. And of course, if multiple policies exist that target the same workload but at different levels of granularity, the proxy will automatically apply the narrowest matching policy: i.e. in our case, this means that a deny policy at the namespace level will always be applied unless an explicit allow exists for the same service at the workload level. The Authorization Policy target is determined by the *metadata/namespace* field and an optional *selector* field to express the eventual specific workload.

Therefore, the next steps were two: the first was to define a DENY AuthorizationPolicy for iam, kafka and tikcketapp namespaces in order to deny by default all traffic entering the various workloads deployed within these namespaces; then, the second was to define multiple ALLOW policies, each targeting a specific workload, so that it could only accept the specified traffic from the defined source.

I first went to consider what should be the normal behaviour of my ticketing application, and then, I identified which service pairs needed to communicate in order to complete tasks and correctly implement the application's functionality. First of all, each microservice (login, catalogue, traveller and payment) must be externally exposed to provide the application's web services, but can only be contacted on its specific exposed HTTP port by the ingress gateway, which will act as an intermediary between the traffic coming from outside and the cluster services: no other entity outside or inside the cluster must be able to contact the four microservices. In this way, imagine in a realistic case that a security node is placed just before the gateway (e.g. a WAF), or between the gateway and the services, the only connections received will have been appropriately security checked, and thus verified. Then, each of the 4 microservices requires access to its own database and thus opens connections only to the specific postgres service for which it is intended. At the same time, Catalogue, Traveler and Payment Service must connect to the Kafka message broker service in order to exchange requests reliably whenever the Catalogue service receives a POST /shop request. Finally, the Kafka service requires, in order to function, to connect to the Zookeeper service, which resides in the same namespace. Figure 5.2 in Chapter 4, shows what has just been described in words.

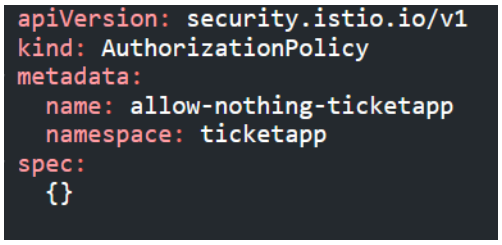
Once I had a complete picture of the communications between the various microservices in the mesh, I moved on to define three namespace-wide *Deny-By-Default* policies (one for each namespace in my application). This will ensure that unless otherwise specified, all traffic received by services within that namespace will be automatically denied by the proxy if it does not match any explicit allow policy.

Figure 6.3 shows the policy specifically used for the ticketapp namespace, but the others are pretty much the same, with only a different target namespace.

Each Envoy proxy runs an authorization engine that authorizes requests at runtime, and whenever a request comes to the proxy, the authorization engine evaluates the request context against the current authorization policies defined, and returns the authorization result, either ALLOW or DENY.

In general, the *selector* field allows you to express the scope to which the policy applies. As can be seen, in this case the policy has no *selector* field to help specify the scope of the policy, and will therefore automatically apply the policy to any workload in the ticketapp namespace. The *spec* field of the policy has the empty value , and therefore the *to,from* and *when* fields are also not specified to indicate any boundary conditions under which the policy is to be applied, such as a specific source, or a certain ports. This means that no traffic is allowed, effectively denying all requests.

Next, I went on to define the specific *ALLOW* policies for each individual pair of services that need to be able to interact and communicate with some specific sources. Figure 6.4 shows the policy required for the Kafka service to be contacted by the 3 microservices Catalogue, Traveler and Payment. The *spec.selector.matchLabels.service* field is used to specify the target service to which the policy has to be applied (the Kafka service in the specific case of the image, or to be more accurate, the workload labeled with *app: kafka* in the Kubernetes resource defined before). The *principal* subfield of the *from.source* field, specifies to the server-side proxy that the identity extracted from the mTLS peer authentication (i.e., the client Service Account) must be used to



```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: allow-nothing-ticketapp
  namespace: ticketapp
spec:
  {}
```

Figure 6.3: Deny-by-default Policy applied to Ticketapp namespace

authorize the request: if, upon successful authentication, the Service Account contained in the source certificate does not match one of those listed in that field, the request will be denied as an essential condition is missing. This will therefore allow pods running Kafka to be accessed only by one of the 3 identities corresponding to the 3 services on port 9092, and by no one else, even if the latter authenticates correctly by presenting a valid certificate but with a different contained identity. Note that in the event that the proxy server is mistakenly set to mTLS DISABLE mode (or in any case PERMISSIVE mode, but plaintext traffic is nevertheless received), mTLS authentication will be absent and consequently it will not be possible to extract and use any principals fields to authorise the traffic: the absence of an extracted principal will in any case lead the proxy to deny the connection since it is unable to verify the condition on the source defined in the policy. Thus, the possible absence of server-side authentication policies will not prevent the proxy from applying authorization policies. The *to.operation.port* field also specifies another boundary condition, namely, on which specific port the traffic must be directed: if the destination port is different from this, the traffic will automatically be denied.

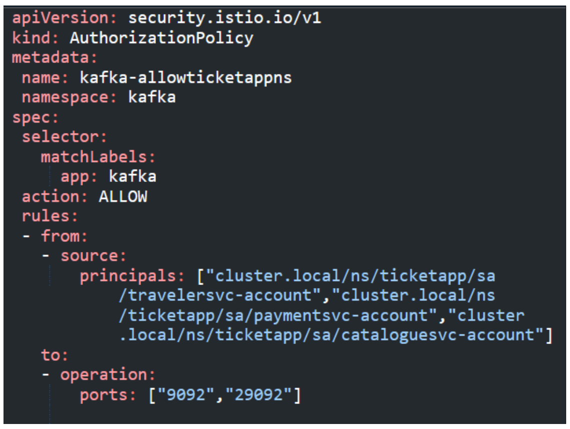
Figure 6.5 shows the policy required for the Zookeeper service in order to be contacted by Kafka: the only source identity from which it can be contacted on port 2181 will therefore be Kafka's service account. All other traffic directed to the same pod, but on different ports or from workloads that do not have that service account and a valid certificate, will be denied (even if the identity is the allowed one).

Figure 6.6 instead, shows the policy that had to be defined for the Postgres service specifically used by the catalogue service, so that only the latter can connect to its database, and no others: a similar policy was also applied to the postgres instances of the Traveler, Payment and Login services to be accessed only by the corresponding microservices. Finally, the figure 6.7 shows the allow policy required by the catalogue service in order to be contacted from outside, i.e. from the Ingress Gateway: the same policy, with different target, identity and port, was also applied to the other three microservices.

6.1.2 End-User Authentication And Authorization

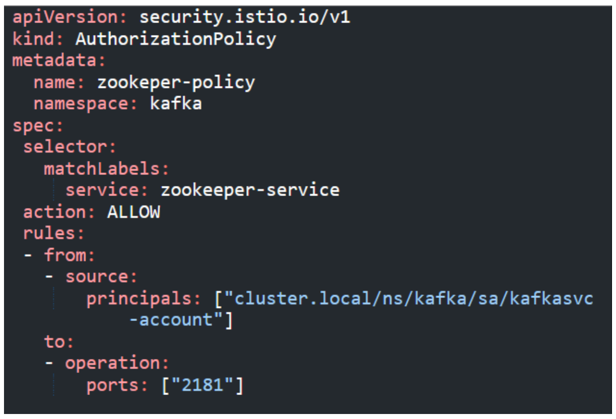
As already mentioned in Chapter 4 and subsection 6.1.1 of Chapter 6, for each authorization policy, Istio provides flexibility by allowing a list of rules to be defined that match the request, using the *rules* field: a policy match occurs when at least one of these rules matches the request. In this way, I can easily make explicit several conditions under which a request can be granted in a single policy. A *rules* field without any rules implies that any request is automatically matched. In order to match a specific request, each rule allows three optional subconditions to be expressed trough the following subfields:

- **from.source**: this field allows to specify the source of a request. If not set, any source is



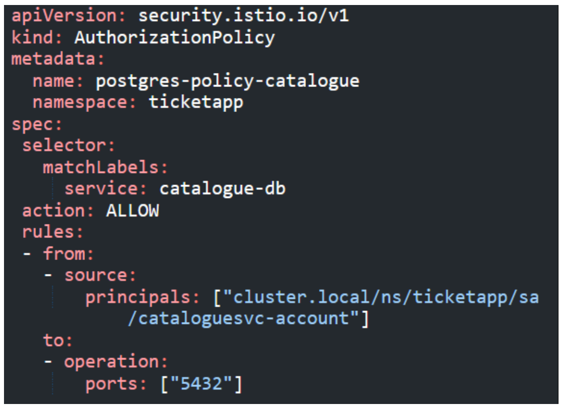
```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: kafka-allowticketappns
 namespace: kafka
spec:
 selector:
   matchLabels:
     app: kafka
 action: ALLOW
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/ticketapp/sa
           /travelersvc-account","cluster.local/ns
           /ticketapp/sa/paymentsvc-account","cluster
           .local/ns/ticketapp/sa/cataloguesvc-account"]
   to:
   - operation:
       ports: ["9092","29092"]
```

Figure 6.4: Allow Policy For Kafka Service



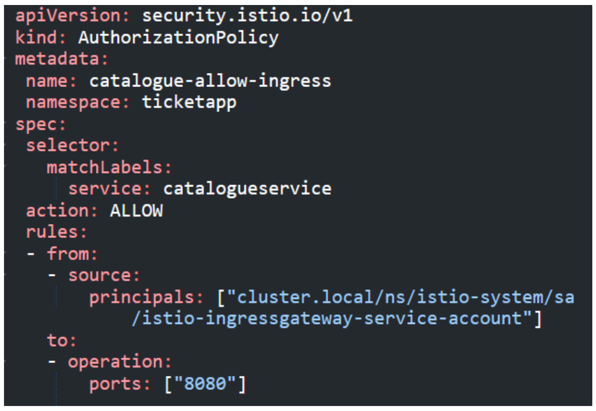
```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: zookeper-policy
  namespace: kafka
spec:
 selector:
   matchLabels:
     service: zookeeper-service
 action: ALLOW
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/kafka/sa/kafkasvc
           -account"]
   to:
   - operation:
       ports: ["2181"]
```

Figure 6.5: Allow Policy For Zookeeper Service



```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: postgres-policy-catalogue
  namespace: ticketapp
spec:
 selector:
   matchLabels:
     service: catalogue-db
 action: ALLOW
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/ticketapp/sa
           /cataloguesvc-account"]
   to:
   - operation:
       ports: ["5432"]
```

Figure 6.6: Allow Policy For Catalogue Service Database



```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: catalogue-allow-ingress
 namespace: ticketapp
spec:
 selector:
   matchLabels:
     service: catalogueservice
 action: ALLOW
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/istio-system/sa
           /istio-ingressgateway-service-account"]
   to:
   - operation:
       ports: ["8080"]
```

Figure 6.7: Allow Policy For Catalogue Service

allowed. You can express three main types of source, which are:

1. a list of peer identities derived from the X.509 certificate using the *from.source.principals* field. You can also express a list of negative matches of peer identities through the function named *from.source.notPrincipals*, which will involve that any principal not matching the specified ones, will match the rule.
2. a list of request identities derived from a validated JWT inside the request, using the *from.source.requestPrincipals* field. You can also express a list of negative matches of request identities through *from.source.notRequestPrincipals*, which will involve that any request principal extracted from a valid JWT not matching the specified ones, will match the rule.
3. a list of IP blocks, derived from the source address of the IP packet, using the *from.source.ipBlocks* field. In this case you can express a single IP address or a CIDR. Also a list of negative matches of IP blocks can be used using the *from.source.notIpBlocks*

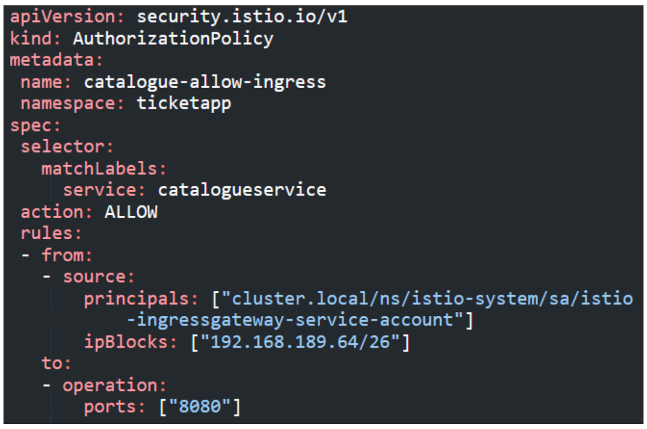
field.

- **to.operation**: this field allows to specify the operations of a request. If not set, any operation is allowed. You can express three main types of operation, which are:
    1. a list of hosts as specified in the HTTP request, trough the *hosts* field. Also a list of negative matches can be expressed trough the *notHosts* field.
    2. a list of ports as specified in the connection, using the *ports* field. Also a list of negative ports can be expressed trough the *notPorts* field.
    3. a list of methods as specified in the HTTP request, using the *methods* field. Also a list of negative methods can be expressed trough the *notMethods* field.
    4. a list of paths as specified in the HTTP request, using the *paths* field. Also a list of negative paths can be expressed trough the *notPaths* field.
- **when**: this field allows to specify a list of additional conditions of a request by expressing a series of attributes that the request should have. If not set, any condition is allowed. You can express conditions using a couple of *key-values* fields for each attribute, in which you can explicit
    1. that a request has a specific HTTP header trough the key value *request.headers*
    2. that a request should (or should not) have a given source and/or destination ip, trough the key value *source.ip* and *destination.ip*
    3. that the valid JWT token contained within the HTTP request, is bound to a specific principal embedded within (trough the *request.auth.principal* field), or that contains a specific claim (trough the *request.auth.claims* field)
    4. and many other less interesting attributes

Fields in the **from.source** section are automatically ANDed together by the proxy, and thus it is possible to specify the source of a request in great detail by specifying both the IP and the identity of the source peer, and also the specific identity to be contained in the JWT (the latter condition, however, can only be applied in the case of HTTP traffic). This would allow us to provide greater security for our workloads, guaranteeing source authorisation on several levels: network level, transport level and application level.

For example, Kubernetes allows labels to be added to cluster nodes, and then allows Pods to be targeted for scheduling on specific nodes or groups of nodes: this functionality can be used to ensure that specific Pods are only executed on nodes with certain isolation, security or regulatory properties. In cases such as these, knowing that a certain workload can only be scheduled on a given node, one can for instance redefine the authorisation policies defined in section 6.1.1 by also specifying the block of IP addresses that has been assigned to the Pods of a specific node, thus achieving a double security check based on both source IP and digital identity (this would for instance also prevent a certain certificate from being stolen and used by different IP addresses). The same can be done in the case where a certain resource deployed in the kubernetes cluster is to be accessed only from within a certain IP subnet or a host with a static IP. A possible example is shown in figure 6.8: the policy provided just before in picture 6.7 has been redefined, specifying the IP addresses actually allowed by the Ingress Gateway pod when contacting the Catalogue service providing a valid certificate and service account within it. If even one of these two conditions miss, the connection is denied.

Also, for what concerning the **to.operation** section, the fields expressed in that section are ANDed together, and thus it is possible to specify the operation of a specific request in great



```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: catalogue-allow-ingress
 namespace: ticketapp
spec:
 selector:
   matchLabels:
     service: catalogueservice
 action: ALLOW
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/istio-system/sa/istio
           -ingressgateway-service-account"]
       ipBlocks: ["192.168.189.64/26"]
   to:
   - operation:
       ports: ["8080"]
```

Figure 6.8: Allow Policy for Catalogue Service with a specific source identity and IP address

detail by specifying port, hostname, path and method of the request. Unfortunately, the last three can only be used in the case of HTTP traffic, thus allowing policies to be more refined for this type of traffic, and only the specific port to be used in the case of simple TCP traffic, as seen in the 6.6 and 6.5 images for kafka, zookeeper and postgres traffic.

However, another security capability of Istio is the fact that it is also able to implement end-user authentication mechanisms based on the single HTTP request, exploiting JWTs that validate the user's login instead of X.509 certificates. This is possible by applying to the API server a .yaml file containing a RequestAuthentication Kubernetes resource.

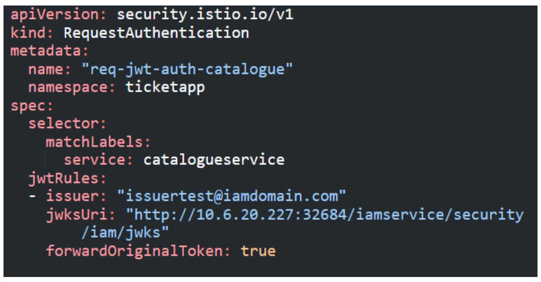
Thus, I first went on to define authentication policies for the individual HTTP requests of my microservices. In fact, as seen in Chapter 5, each microservice has endpoints that require prior authentication to the IAM server in order to be accessed, i.e. they require a valid JSON Web Token (JWT) in the request's authorization header in order to be accessed since they imply access to a specific user's personal data.

In particular, the Login service presents only one authenticated endpoint, *PUT /admin/users/<userID>*, and all the others are freely accessible as they serve to guarantee registration, validation and login operations. Payment and Traveler services, on the other hand, present only authenticated endpoints. Finally, the Catalogue service has all authenticated endpoints except *GET /tickets*, which must also be freely accessible by non-authenticated users in order to be able to show the catalogue of tickets on sale to the end user.

In order to define a RequestAuthentication policy, three pieces of information must be defined:

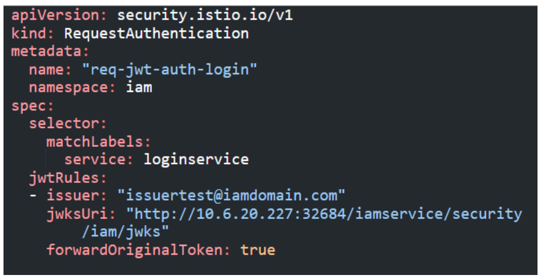
- the location of the token in the request (i.e., the HTTP header in which the JWT to be authenticated is contained)
- the issuer of the request, that is the value of the *iss* field contained in the token.
- the JSON Web public key set (JWKS). It is a set of keys containing the public keys used to verify any JSON Web Token (JWT) issued by the Authorization Server and signed using the RS256 signing algorithm.

I therefore went on to define a RequestAuthentication policy for each individual microservice, specifying these three pieces of information for each one. Figures 6.9 and 6.10 show the ones provided for the Catalogue and Login services, even if the other two are similar.



```
apiVersion: security.istio.io/v1
kind: RequestAuthentication
metadata:
  name: "req-jwt-auth-catalogue"
  namespace: ticketapp
spec:
  selector:
    matchLabels:
      service: catalogueservice
  jwtRules:
  - issuer: "issuertest@iamdomain.com"
    jwksUri: "http://10.6.20.227:32684/iamservice/security
        /iam/jwks"
    forwardOriginalToken: true
```

Figure 6.9: RequestAuthentication for Catalogue Service



```
apiVersion: security.istio.io/v1
kind: RequestAuthentication
metadata:
  name: "req-jwt-auth-login"
  namespace: iam
spec:
  selector:
    matchLabels:
      service: loginservice
  jwtRules:
  - issuer: "issuertest@iamdomain.com"
    jwksUri: "http://10.6.20.227:32684/iamservice/security
        /iam/jwks"
    forwardOriginalToken: true
```

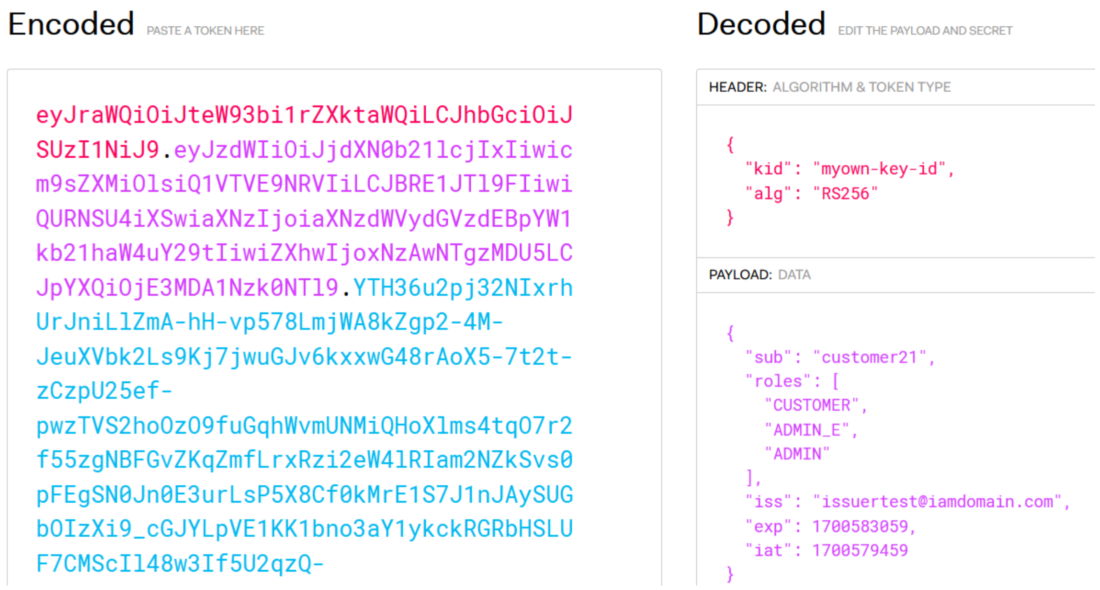
Figure 6.10: RequestAuthentication for Login Service

As can be seen from the images above, the details from which the proxies will retrieve information to verify the JWT must be expressed in the jwtRules field. In particular, a *jwksUri* field has been specified for the JWKS: this represents a URI that will be provided to each proxy so that it is able to download and retrieve the valid JWKS periodically, and subsequently it will be used for JWT validation and verification. In this case, having realised a custom IAM service, I set the URI of the purpose-built server used to retrieve the JWKS, namely */iamservice/security/iam/jwks*. At each successful user login, the Login service returns a digitally signed JWT, the signature of which can be verified using the public key contained in the downloaded JWKS. Furthermore, the JWT generated by the Login service will have the form shown in the figure 6.11 Note that it has five claims, *sub*, *roles*, *iss*, *exp*, *iat* respectively. The *iss* represents the issuer of the token, which in the case of my custom IAM service will always be "issuertest@iamdomain.com", the reason why I set it as a *issuer* field in the policy. The *sub* claim represents the identity of the user, in this specific case its own username. Furthermore, since the endpoints also need to receive the JWT in order to be able to extract the user's identity information and manage its data, it was necessary to explicitly tell the proxy to forward the JWT to the application container as well, otherwise by default the Envoy's behaviour would have been to validate the jwt, and then remove it from the request before forwarding it. Unless otherwise expressed through the *fromHeaders* field, by default envoy assumes the token is presented within the Authorization header and tries to extract it from there automatically. Therefore, since in our case it is inserted right there, there was no need to specify that field.

By defining only the request authentication policies, what happens is that Istio verifies the digital signature of the token received in the HTTP request using the JWKS downloaded from the URL, then verifies its authenticity and integrity, and once this first check has been passed, it then verifies that the correct issuer is contained in the *iss* field. If even one of these checks fails, the JWT is considered invalid, and consequently, the request is immediately denied. However, if the HTTP request does not carry any tokens inside to be authenticated and validated, RequestAuthentication alone will fail, and by default will let the request pass anyway. Which is why, once we've applied the RequestAuthentication policies we've seen, we must also ensure that no HTTP requests without tokens are allowed to pass through, and this must be done by forcing AuthorizationPolicies.

In addition to being able to validate the JWTs contained in the headers, and thus verify that they are not expired or compromised, the Envoy proxy is able to extract the information contained in them (the so-called *claims*) to subsequently authorize the request.

Therefore, I then defined a DENY AuthorizationPolicy for each microservice, in which I would



Encoded PASTE A TOKEN HERE

```
eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJ
SUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwic
m9sZXMiOlsiQ1VTVE9NRVIiLCJBRE1JTl9FIiwi
QURNSU4iXSwiaXNzIjoiaXNzdWVydGVzdEBpYW1
kb21haW4uY29tIiwiZXhwIjoxNzAwNTgzMDU5LC
JpYXQiOjE3MDA1Nzk0NTl9.YTH36u2pj32NIxrh
UrJniLlZmA-hH-vp578LmjWA8kZgp2-4M-
JeuXVbk2Ls9Kj7jwuGJv6kxxwG48rAoX5-7t2t-
zCzpU25ef-
pwzTVS2hoOzO9fuGqhWvmUNMiQHoXlms4tqO7r2
f55zgNBFGvZKqZmfLrxRzi2eW4lRIam2NZkSvs0
pFEgSN0Jn0E3urLsP5X8Cf0kMrE1S7J1nJAySUG
bOIzXi9_cGJYLpVE1KK1bno3aY1ykckRGRbHSLU
F7CMScIl48w3If5U2qzQ-
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "kid": "myown-key-id",
  "alg": "RS256"
}
```

PAYLOAD: DATA

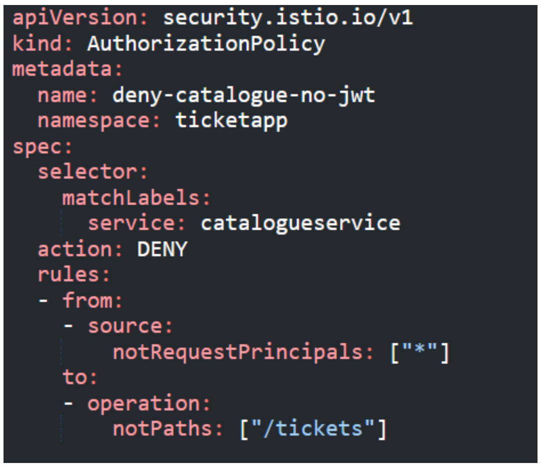
```
{
  "sub": "customer21",
  "roles": [
    "CUSTOMER",
    "ADMIN_E",
    "ADMIN"
  ],
  "iss": "issuertest@iamdomain.com",
  "exp": 1700583059,
  "iat": 1700579459
}
```

Figure 6.11: An example of JWT produced by the Login Service

explicitly deny all traffic that did not contain any claim principals: the *Request Principal* is an attribute of the specific request which is automatically extracted by the proxy from within the received JWT, and which will therefore be absent in the absence of JWT. It corresponds to the *sub* ("subject") claim of the JWT, which is a string that identifies the principal that is the subject of the JWT. The "*" character in *notRequestPrincipals["*"]* serves as a wildcard to match any principals extracted from the JWT, and the fact that it is therefore denied, results in the condition "if no principal has been extracted from the JWT, either because that field is absent, or because the JWT is absent, then deny the request".

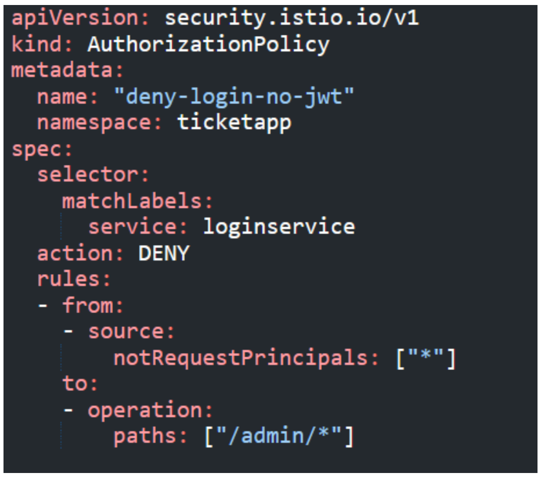
The images 6.12 and 6.13 show the AuthorisationPolicies for the Catalogue and the Login Service. Note that the condition *to.operation.notPaths: ["/tickets"]* has also been expressed for the Catalogue, since the /ticket endpoint is not authenticated, and therefore does not require JWT to access it, and by doing so I guarantee that the policy will not deny requests without JWT directed to this endpoint (but at most will authenticate the JWT if it is present anyway, since the authentication policy remains valid anyway). Similarly, in the case of the Login Service on the other hand, the condition *to.operation.paths: ["/admin/*"]* has been expressed, since the only authenticated endpoint to which the JWT's necessary presence condition is to be applied is the /admin/users/<userID> endpoint. All its other endpoints, instead, do not require JWT to access them, and therefore no *to.operation.paths* or *to.operation.notPaths* condition was required this time, but just the *notRequestPrincipals["*"]* one.

After that, I moved on to exploit the jwtRules offered by the AuthorizationPolicy Istio to authorize access to all endpoints of my services that required administrative privileges and would, therefore also be more sensitive in a realistic scenario. I then defined an additional authorization policy for all endpoints requiring the Admin role, exploiting the *roles* claims contained in each issued JWT, extracted upon successful JWT validation: this claim contains the list of roles of each principal, and only those who possess the "ADMIN" role (as shown in Figure 6.11) are considered as such and can access these privileged endpoints.



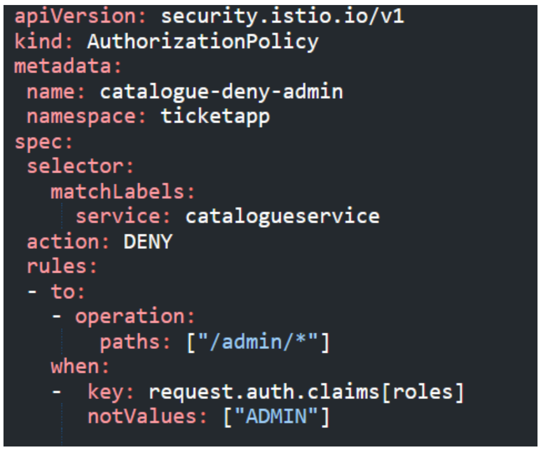
```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: deny-catalogue-no-jwt
  namespace: ticketapp
spec:
  selector:
    matchLabels:
      service: catalogueservice
  action: DENY
  rules:
  - from:
    - source:
        notRequestPrincipals: ["*"]
    to:
    - operation:
        notPaths: ["/tickets"]
```

Figure 6.12: JWT Authorization Policy for Catalogue Service



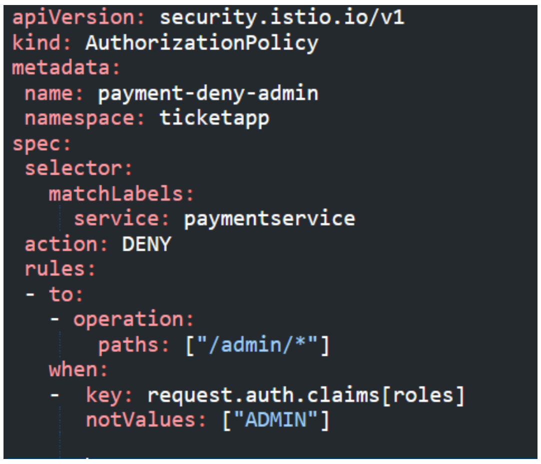
```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: "deny-login-no-jwt"
  namespace: ticketapp
spec:
  selector:
    matchLabels:
      service: loginservice
  action: DENY
  rules:
  - from:
    - source:
        notRequestPrincipals: ["*"]
    to:
    - operation:
        paths: ["/admin/*"]
```

Figure 6.13: JWT Authorization Policy for Login Service



```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: catalogue-deny-admin
 namespace: ticketapp
spec:
 selector:
   matchLabels:
     service: catalogueservice
 action: DENY
 rules:
 - to:
   - operation:
       paths: ["/admin/*"]
   when:
   -  key: request.auth.claims[roles]
      notValues: ["ADMIN"]
```

Figure 6.14: Admin Authorization Policy for Catalogue Service



```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: payment-deny-admin
 namespace: ticketapp
spec:
 selector:
   matchLabels:
     service: paymentservice
 action: DENY
 rules:
 - to:
   - operation:
       paths: ["/admin/*"]
   when:
   -  key: request.auth.claims[roles]
      notValues: ["ADMIN"]
```

Figure 6.15: Admin Authorization Policy for Payment Service

I then went on to define an additional DENY Authorization Policy for each microservice, specifying as conditions that all traffic directed to endpoints beginning with the prefix "/admin" i.e. *to.operation.paths: ["/admin/*"]*, should be denied if it does not have the value "ADMIN" in the claim roles of the JWT, i.e. *when request.auth.claims[roles] notValues["ADMIN"]*, as also shown in Figures 6.14 and 6.15.

6.1.3 Ingress And Egress traffic

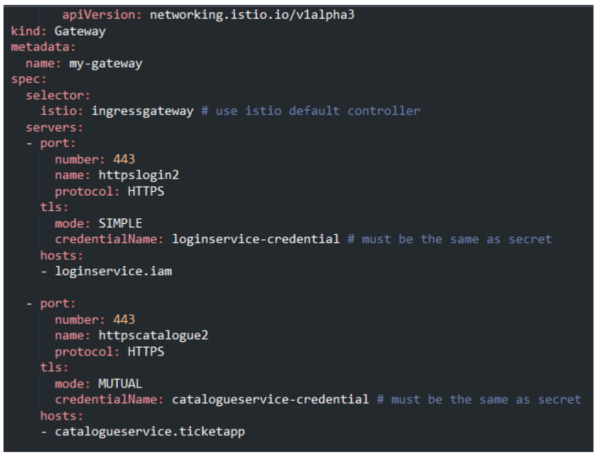
For incoming traffic to the cluster, the Istio Ingress Gateway implements the same functionality as the proxies. It is able to act as a TLS Gateway for HTTP traffic coming from outside, also exposing an HTTPS port and possibly performing Peer Authentication and Authorization leveraging upon the TLS Client Certificate at the edge.To test its functionality, I wrote a YAML file to overwrite the ingress-gateway configuration with my own custom one, requiring MUTUAL TLS requests for each service of the Mesh. In Figure 6.16, there is an example of how I configured the Gateway for accepting the traffic directed to Login service and Catalogue service. Note that in general for each specific service that runs in my mesh I can configure the gateway to require TLS or not, and if required, I can specify whether I want it simple or mutual, as I have done here for example: the Login service requires it simple, and therefore even if the client does not present any certificate the request will be accepted, instead the Catalogue mutual, rejecting clients without a certificate or with an invalid certificate. Thus, for the Ingress Gateway, there is no explicit peer authentication policy, but simply configure the exposed HTTPS port to accept only clients with a valid certificate via the mutual option.

Other features provided by Istio ingress gateway are Request Authentication and Authorization by means of the JWT attached to the incoming HTTP request: it can verify the authenticity and integrity of the JWT attached to the incoming HTTP request, and then perform End-user Authorization. If either authentication or authorization phase fails, we are able to reject the request before it reaches the destination service.

In order to test this concept, I configured a Request Authentication policy with Istio RequestAuthentication resource as Figure 6.17, and I applied it to the Ingress Gateway. Then I configured also two Authorization policy, one for let requests directed to some specific endpoints pass without requiring any attached JWT (Figure 6.18, and another one to strictly require the ADMIN role in the JWT claim whenever the /admin endpoints of the various services are required. In this way, except for a few specific ones, all endpoints will require a valid JWT, i.e. that the user has previously logged in (the login service is obviously excluded from this condition for obvious reasons), and all those that have an invalid or absent JWT will be rejected already by the ingress gateway. Finally, if endpoints are reserved for admins, all requests to them from users without that role will be refused, as also seen in Chapter 6.1.2.

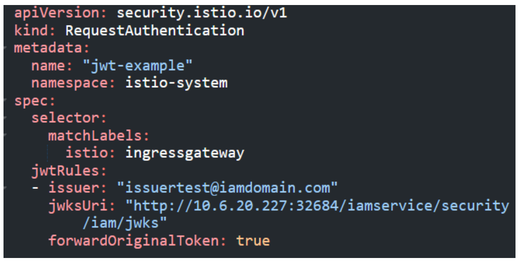
With regard to outgoing traffic from the service mesh and the services within it, all outgoing traffic from an Istio-enabled pod is intercepted by its sidecar proxy, which will handle it correctly. Accessibility to URLs outside the cluster depends on the proxy configuration, and by default, Istio configures the Envoy proxy to pass through requests for unknown services, and thus all connections towards external HTTP and HTTPS services from applications inside the mesh, and in general any traffic towards unknown destinations, are implicitly allowed. This behaviour can be overridden at a Sidecar level by setting the OutboundTrafficPolicy to REGISTRY ONLY at installation time. Istio has an internal service registry containing several Service Entries, each one with information about workloads within the mesh (its DNS name, clusterIP, IP address, etc). Services that are not defined in Istio's internal service registry, are considered unknown: this is the case of the services external to the cluster, i.e. on the Internet.

If the option is set to REGISTRY ONLY, then the Istio proxy automatically will block any host without an HTTP service or service entry defined within the registry, preventing anomalous outgoing connections. However, this is not a strong security measure, since if the proxy is by-passed, the job is done. While this is useful to prevent accidental dependencies, if you want to secure egress traffic, and enforce all outbound traffic goes through a specific proxy, maybe performing strong security checks, you should instead rely on an Egress Gateway. When combined



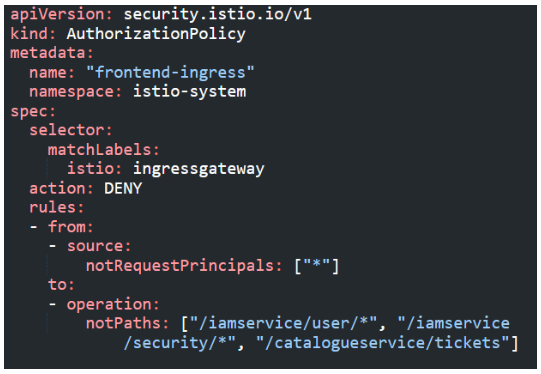
```yaml
    apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: my-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 443
      name: httpslogin2
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: loginservice-credential # must be the same as secret
    hosts:
    - loginservice.iam

  - port:
      number: 443
      name: httpscatalogue2
      protocol: HTTPS
    tls:
      mode: MUTUAL
      credentialName: catalogueservice-credential # must be the same as secret
    hosts:
    - catalogueservice.ticketapp
```

Figure 6.16: Ingress Gateway Configuration



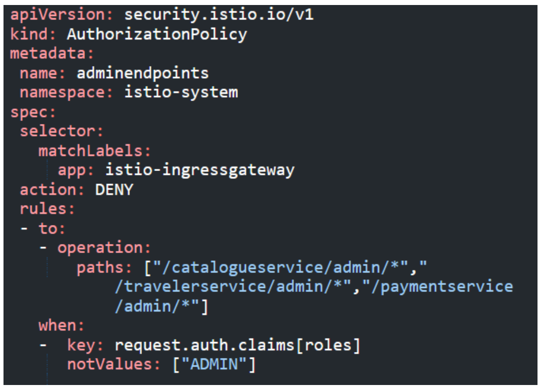
```yaml
apiVersion: security.istio.io/v1
kind: RequestAuthentication
metadata:
  name: "jwt-example"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  jwtRules:
  - issuer: "issuertest@iamdomain.com"
    jwksUri: "http://10.6.20.227:32684/iamservice/security
        /iam/jwks"
    forwardOriginalToken: true
```

Figure 6.17: Ingress Gateway Request Authentication



```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: "frontend-ingress"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  action: DENY
  rules:
  - from:
    - source:
        notRequestPrincipals: ["*"]
    to:
    - operation:
        notPaths: ["/iamservice/user/*", "/iamservice
            /security/*", "/catalogueservice/tickets"]
```

Figure 6.18: Ingress Gateway Request Authorization



```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: adminendpoints
 namespace: istio-system
spec:
 selector:
   matchLabels:
     app: istio-ingressgateway
 action: DENY
 rules:
 - to:
   - operation:
       paths: ["/catalogueservice/admin/*","
           /travelerservice/admin/*","/paymentservice
           /admin/*"]
   when:
   - key: request.auth.claims[roles]
     notValues: ["ADMIN"]
```

Figure 6.19: Ingress Gateway Admin Authorization

with a Network Policy, you can enforce all traffic, or some subset, that goes through the egress gateway. This ensures that even if a client accidentally or maliciously bypasses their sidecar, the request will be blocked.

I then went on to reinstall the service mesh using the REGISTRY ONLY configuration, and then to appropriately configure the routing of outgoing traffic, since we have already deployed our egress gateway, but currently outgoing traffic does not pass through there. In particular, to test access to external services via the egress gateway, I used the service "www.google.com" as an example. assuming that only access to the latter was required from within the mesh services. Thus, It has been configured through a Gateway resource (Figure 6.21) and, in order to define the exiting routes, it also required a Destination Rule and a Virtual Service. The virtual service was used to configure two routes, the first from inside the mesh to the exit gateway and the second from the gateway to the external service, defined via the host and destination port fields, as shown in Figure 6.20: it thus will direct the traffic from the sidecars to the egress gateway and then from the egress gateway to the external service. The entries of this Istio's internal registry can be defined with the Service Entry resource, by specifying the host, the port, the type of protocol, then also whether the service is internal or external to the mesh, and finally how the destination host must be resolved (often using the DNS service). Therefore, also a Service Entry has been created for the Google service so that this outgoing communication will not be blocked by Istio (Figure 6.22). With those resources, we created an exit gateway for google.com, port 443: to route multiple hosts through an exit gateway, you can include a list of hosts, or use * to match them all, in the gateway.

6.2 Palo Alto CN-Series Containerized Firewall

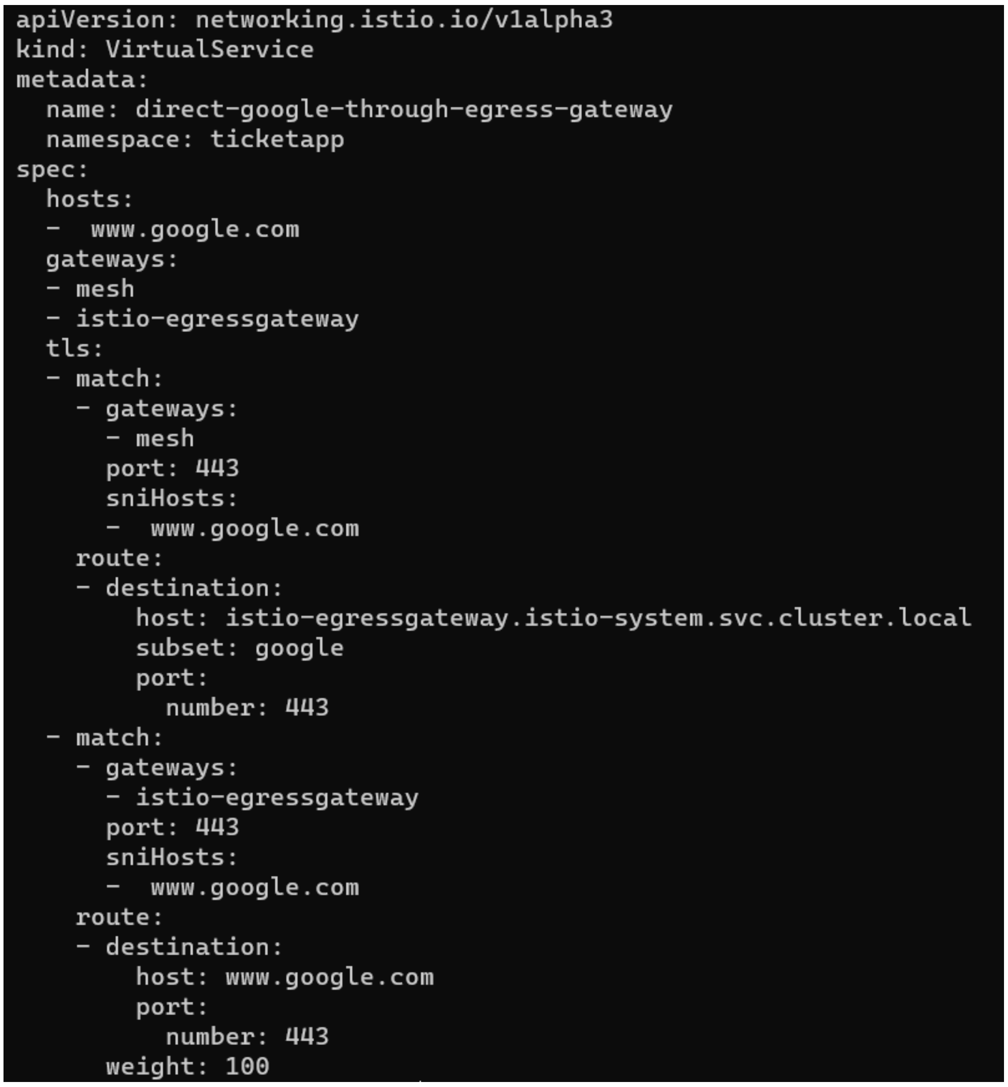
6.2.1 CN-Series Components Deployment

The first step in testing the containerised firewall solution was to deploy Panorama within our cluster. A virtual machine ready to run Panorama was provided by Palo Alto and installed with a suitable resource configuration to function properly. It provides a centralised management platform for Palo Alto Networks firewalls, including the CN Series. Panorama allows administrators to manage multiple CN-Series firewalls from one centralised interface, including configuration management, policy implementation and monitoring. It can be used to define expected security policies and their enforcement, pushing them to management pods to configure NGFW pods. In addition, Panorama is able to collect logs from all managed CN-Series firewalls, providing centralised log storage and analysis capabilities. This provides complete visibility of network traffic and security events across the entire environment, while also offering reporting and alerting capabilities.

I then went on to download and install the Kubernetes plugin on Panorama, using a suitable version that also matched the version of Kubernetes installed on our cluster: it is necessary to configure the IP address and port of the K8s API server running, so that Panorama is able to monitor through it the creation and destruction of K8s resources, as well as any changes occurring within the cluster. Furthermore, since the API server requires authenticated communication, a ServiceAccount was specially created for Panorama to use each time it contacts the API server. The corresponding ServiceAccount will, of course, have permission to read the existing resource configuration and deployment details. In this way, Panorama is now able to monitor the cluster and thus collect information about existing Kubernetes objects, such as pods, services, deployment and associated identifying attributes, and this will consequently allow us to create context-aware policies. The time interval used to periodically query the API server and retrieve information has been set to 30 seconds, but this can be changed.

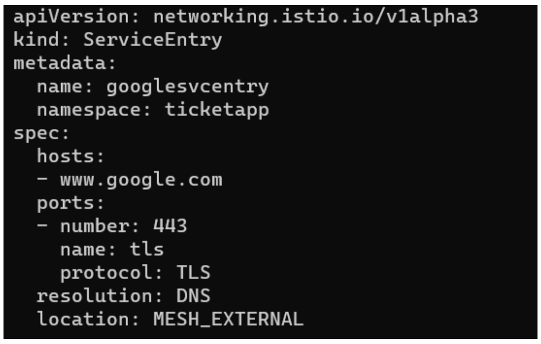
Next, I downloaded the CN-series deployment files from Palo Alto Networks' GitHub repository and, by customising and applying these YAML files to the API server, I inserted the PN-CNI into each node's CNI chain, then deployed the management and firewall pods within the cluster: NGFWs, management pods, and Panorama pods were configured through these files to communicate properly with each other via IPsec tunnels, ensuring secure communication. Figure 6.23 shows a part of the YAML file used for the NGFW deployment.

The DaemonSet mode was chosen to distribute the firewalls: consequently, since our cluster consists of two worker nodes, two NGFWs were distributed within the cluster, one for each node. For fault-tolerance issues, however, two management pods were also deployed, guaranteeing no service interruptions in the event of a failure of one. As you can see from Figure 6.24, all the cn-series pods are deployed within the kube-system namespace, since they should be granted significant permissions.



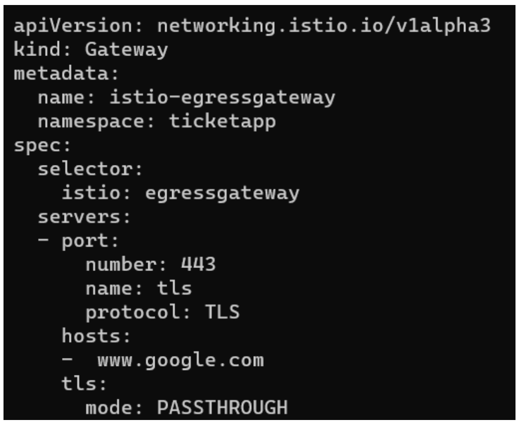
```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: direct-google-through-egress-gateway
  namespace: ticketapp
spec:
  hosts:
  -  www.google.com
  gateways:
  - mesh
  - istio-egressgateway
  tls:
  - match:
    - gateways:
      - mesh
      port: 443
      sniHosts:
      -  www.google.com
    route:
    - destination:
        host: istio-egressgateway.istio-system.svc.cluster.local
        subset: google
        port:
          number: 443
  - match:
    - gateways:
      - istio-egressgateway
      port: 443
      sniHosts:
      -  www.google.com
    route:
    - destination:
        host: www.google.com
        port:
          number: 443
      weight: 100
```

Figure 6.20: Virtual Service Resource



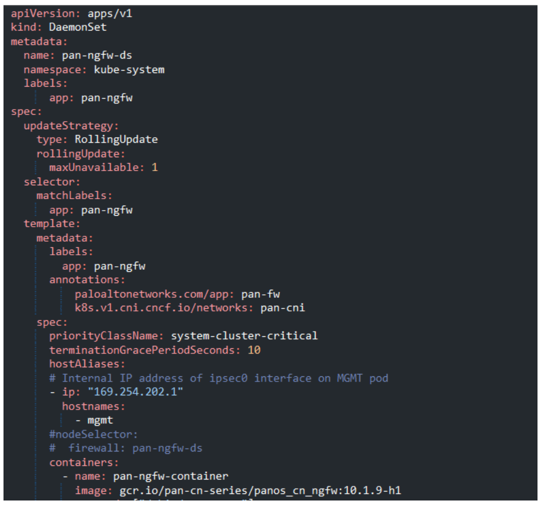
```yaml
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: googlesvcentry
  namespace: ticketapp
spec:
  hosts:
  - www.google.com
  ports:
  - number: 443
    name: tls
    protocol: TLS
  resolution: DNS
  location: MESH_EXTERNAL
```

Figure 6.21: Service Entry Resource



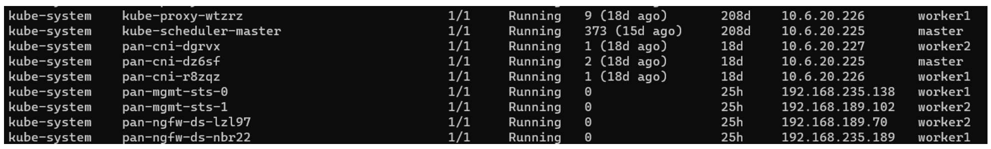
```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-egressgateway
  namespace: ticketapp
spec:
  selector:
    istio: egressgateway
  servers:
  - port:
      number: 443
      name: tls
      protocol: TLS
    hosts:
    -  www.google.com
    tls:
      mode: PASSTHROUGH
```

Figure 6.22: Gateway Resource



```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: pan-ngfw-ds
  namespace: kube-system
  labels:
    app: pan-ngfw
spec:
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  selector:
    matchLabels:
      app: pan-ngfw
  template:
    metadata:
      labels:
        app: pan-ngfw
      annotations:
          paloaltonetworks.com/app: pan-fw
          k8s.v1.cni.cncf.io/networks: pan-cni
    spec:
      priorityClassName: system-cluster-critical
      terminationGracePeriodSeconds: 10
      hostAliases:
      # Internal IP address of ipsec0 interface on MGMT pod
      - ip: "169.254.202.1"
        hostnames:
          - mgmt
      #nodeSelector:
      #  firewall: pan-ngfw-ds
      containers:
        - name: pan-ngfw-container
          image: gcr.io/pan-cn-series/panos_cn_ngfw:10.1.9-h1
```

Figure 6.23: NGFW YAML Configuration File



```
kube-system    kube-proxy-wtzrz                  1/1     Running   9 (18d ago)      208d    10.6.20.226       worker1
kube-system    kube-scheduler-master             1/1     Running   373 (15d ago)    208d    10.6.20.225       master
kube-system    pan-cni-dgrvx                     1/1     Running   1 (18d ago)      18d     10.6.20.227       worker2
kube-system    pan-cni-dz6sf                     1/1     Running   2 (18d ago)      18d     10.6.20.225       master
kube-system    pan-cni-r8zqz                     1/1     Running   1 (18d ago)      18d     10.6.20.226       worker1
kube-system    pan-mgmt-sts-0                    1/1     Running   0                25h     192.168.235.138   worker1
kube-system    pan-mgmt-sts-1                    1/1     Running   0                25h     192.168.189.102   worker2
kube-system    pan-ngfw-ds-lzl97                 1/1     Running   0                25h     192.168.189.70    worker2
kube-system    pan-ngfw-ds-nbr22                 1/1     Running   0                25h     192.168.235.189   worker1
```

Figure 6.24: CN-Series core blocks deployed in the cluster

A different pan-cni pod is distributed on each node in the cluster to apply the CNI cn-series specification, while a different NGFW and MGMT pod is distributed on each worker node: the master node does not require them, as it does not host and run applications on top, but only the control plane pods.

However, in order for firewall pods to intercept incoming or outgoing traffic from a given application pod, the application yaml or corresponding namespace had to be annotated using the label *paloaltonetworks.com/firewall=pan-fw*. I opted for the latter, since this implied that each newly created pod in the specified namespace would be directly connected to the firewall and, in order to guarantee zero-trust principles, each workload in the cluster's namespace had to be protected via a PEP. I then annotated the various namespaces of my applications (i.e. ticketapp, iam and kafka) using the command **kubectl annotate namespace <namespace-name> paloaltonetworks.com/firewall=pan-fw**. Since the DaemonSet mode has been adopted, the CNI is then configured to connect the network interface of each newly created pod within these annotated namespaces to the firewall via a virtual wire, avoiding any kind of bypass: in this way, we are quite sure that whenever traffic exits or enters the pod, it is because the firewall lets it pass after an appropriate authorisation policy has been applied. Of course, in this mode, pods scheduled on a specific node can only be connected to the NGFW instance running on that node. This implies that a maximum of 30 application pods can be connected to each firewall (since it only has 30 pairs of vwire interfaces), but since I only deployed a few test pods, this was sufficient in my case.

Log Forwarding was then enabled and configured for each NGFW pod to automatically send logs on allowed or denied traffic to Panorama, which will collect and store them. Panorama's

K8s plugin continuously monitors the API server and has been configured to automatically create tags for the following Kubernetes objects:

- Pod Classes: ReplicaSets, DaemonSets, StatefulSets
- Service Types: ClusterIP, NodePort, LoadBalancer
- Service Objects: port, targetPort, nodePort, and pod interfaces

Thus, finally, the Kubernetes plugin was configured to dynamically send the created tags to the management pods and, subsequently, to configure the firewall and its policies using the source and destination tags of each rule by exploiting the IP-tag address mapping.

6.2.2 Microsegmentation Trough Firewall Security Policies

In order to test the security capabilities of the containerised firewall, as I also did in the case of Istio, I first considered how the microservices interact with each other within my ticketing application, and thus which services are exposed by each workload, and which pair of workloads actually requires to communicate and use which specific service.

Four microservices (login, catalogue, traveler and payment) must be exposed outside the cluster to provide the application's web services, but they can only be contacted on their specific HTTP port, which must be exposed via the Ingress Gateway, which will act as a reverse proxy routing traffic from outside to the specific workload. Thus, the first consideration is that the Istio service mesh has been uninstalled, but its Ingress Gateway has been retained to ensure public exposure of the web microservices, acting as the only point of contact between the cluster and the rest; the second is that only the gateway must be able to contact the four workloads: no other entity must be allowed. Naturally, then, each of the four microservices requires access to its own database, and thus to open connections only to the specific postgres service for which it is intended. Catalogue, Traveler and Payment Services must be able to use the Kafka message broker service to reliably exchange requests whenever the Catalogue service service receives a POST /shop request: therefore, traffic between these three workloads and the Kafka service must be allowed. Finally, the Kafka service requires to connect to the Zookeeper service in order to function. Taking this into account, to micro-segment traffic between Kubernetes pods, a whitelisting approach was considered and implemented: this means that to protect the cluster's connections and requests, a default-deny authorisation model was adopted which denies all requests by default and only accepts those that are explicitly allowed.

Since the goal is to achieve Zero Trust through pod firewalls, all three ticketing app namespaces have been annotated with the label textitpaloaltonetworks. com/firewall=pan-fw, and this will ensure that every time a pod starts or restarts within these namespaces, the PAN-CNI will connect its network interface to the pod firewall running on the same worker node (this is true for DaemonSet mode, while for Service mode a VxLAN tunnel is adopted), and traffic entering or leaving that pod will necessarily first have to pass through the pod firewall, be inspected, and then only if authorised will it be forwarded to the correct destination. This is possible through virtual wires: each firewall has 30 pairs of virtual interfaces, where each pair consists of one interface connected to the pod, and the other to the K8s overlay network. The one connected to the pod is labelled 'trust', while the other is labelled 'untrust', clearly decoupling what is the implicit trust zone from the rest. The firewall connects these two interfaces internally, ensuring that anything that enters one interface and is authorised by the PEP can only reach that which is connected to the other interface linked to the first.

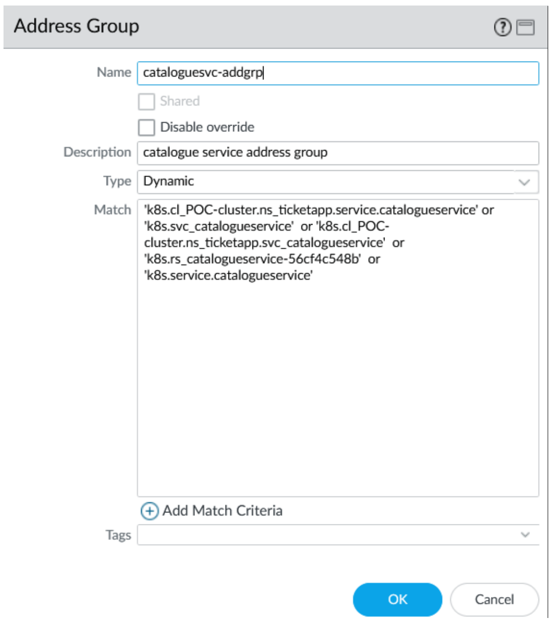
As also reported by NIST, true Zero Trust can only be implemented by ensuring that a security perimeter is placed around every workload (or at least every sensitive load). Therefore, in a real-world scenario, all namespaces within the cluster should be annotated and their pods protected by the firewall: however, in this case, the Service mode implementation is suggested, since the DaemonSet one would not be scalable with a large number of pods per node (it can only protect at most 30 pods per node in DaemonSet mode).

However, with regard to the pods already present in the namespaces annotated prior to the deployment of the PAN-CNI and the firewall pods, traffic interception cannot take place, as their network interfaces can only be appropriately configured for redirection at the time of deployment, thus ensuring that no one can bypass the firewall. Therefore, it was necessary to restart the various workloads of the ticketing application to actually place the PEP in front of each of them.

The second step was to create the so-called **Dynamic Address Group** (DAG). As already mentioned in the previous chapters, the dynamism of the Kubernetes environment also implies a continuous spin up and down of pods, and consequently a never-consistent and stable IP address for each pod. **DAGs** help manage these ephemeral entities by exploiting the *IP-address-to-tag mapping* function provided by the K8s plugin for pods, nodes, namespaces and services within the cluster. In fact, the plugin continuously polls and monitors the API server, retrieving information on the created K8s entities and automatically generating a set of labels (or tags) using a specific hierarchical format and mapping these labels to a set of IP addresses corresponding to the entities associated with these K8s resources. For example:

- For each existing *Namespace*, a label with the following format is created: **k8s.cl_<cluster-name>.ns_<namespace>**. The plugin retrieves the information about the IP addresses of the existing pods within that specific namespace, and map them to that label.
- For each existing *DaemonSet*, a label with the following format is created: **k8s.cl_<cluster-name>.ns_<namespace>.ds_<pod-name>**. The plugin retrieves the information about the IP addresses of the existing pods which are part of that specific DaemonSet deployed within that specific namespace, and map them to that label.
- For each existing *ReplicaSet*, a label with the following format is created: **k8s.cl_<cluster-name>.n_<namespace>.rs_<pod-name>**. The plugin retrieves the information about the IP addresses of the existing pods, which are part of that specific ReplicaSet deployed within that specific namespace, and maps them to that label.
- For each existing *StatefulSet*, a label with the following format is created: **k8s.cl_<cluster-name>.ns_<namespace>.ss_<pod-name>**. The plugin retrieves the information about the IP addresses of the existing pods which are part of that specific StatefulSet deployed within that specific namespace, and map them to that label.
- For each existing *Service*, a label with the following format is created: **k8s.cl_<cluster-name>.ns_<namespace>.svc_<svc-name>**. The plugin retrieves the information about the IP addresses of the existing pods which are part of that specific Service (and also the clusterIP address) deployed within that specific namespace, and map them to that label.
- And many others...

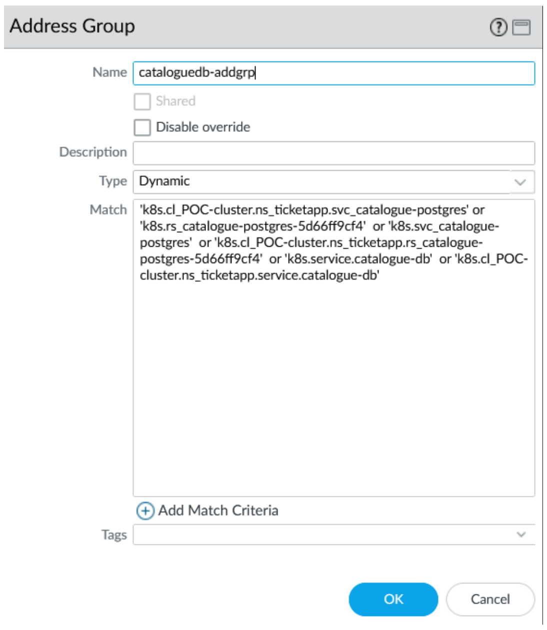
A Dynamic Address Group thus consists of a group of IP addresses that can change dynamically: you simply specify a name for the group and a set of matching criteria to be used for including some specific IP addresses within this group at runtime, and then you can use these DAGs as sources or destinations of your security policies, ensuring greater ease and flexibility in policy definition, while also ensuring that your security policy is always up-to-date, taking into account the dynamic



Address Group

Name	cataloguesvc-addgrp
	Shared
	Disable override
Description	catalogue service address group
Type	Dynamic
Match	'k8s.cl_POC-cluster.ns_ticketapp.service.catalogueservice' or 'k8s.svc_catalogueservice' or 'k8s.cl_POC-cluster.ns_ticketapp.svc_catalogueservice' or 'k8s.rs_catalogueservice-56cf4c548b' or 'k8s.service.catalogueservice'
Tags	

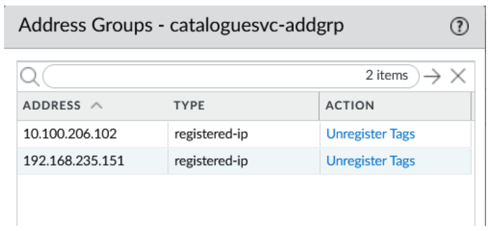
Figure 6.25: DAG for the Catalogue Service



Address Group

Name	cataloguedb-addgrp
	Shared
	Disable override
Description	
Type	Dynamic
Match	'k8s.cl_POC-cluster.ns_ticketapp.svc_catalogue-postgres' or 'k8s.rs_catalogue-postgres-5d66ff9cf4' or 'k8s.svc_catalogue-postgres' or 'k8s.cl_POC-cluster.ns_ticketapp.rs_catalogue-postgres-5d66ff9cf4' or 'k8s.service.catalogue-db' or 'k8s.cl_POC-cluster.ns_ticketapp.service.catalogue-db'
Tags	

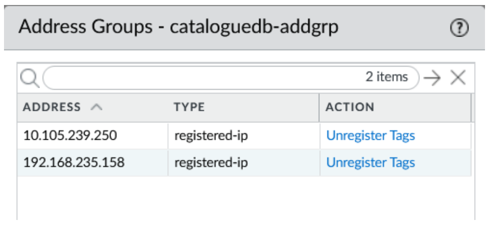
Figure 6.26: DAG for the Catalogue Database Service



Address Groups - cataloguesvc-addgrp

ADDRESS	TYPE	ACTION
10.100.206.102	registered-ip	Unregister Tags
192.168.235.151	registered-ip	Unregister Tags

Figure 6.27: IP addresses for the Catalogue Service DAG



Address Groups - cataloguedb-addgrp

ADDRESS	TYPE	ACTION
10.105.239.250	registered-ip	Unregister Tags
192.168.235.158	registered-ip	Unregister Tags

Figure 6.28: IP addresses for the Catalogue Database Service DAG

changes in the Kubernetes environment and entity IPs. These criteria are expressed via labels created by the Panorama K8s plugin, and this implies that whenever the set of IP addresses associated with that label changes (perhaps because the number of pods scales horizontally and new ones associated with that label are created, or perhaps because a pod has been terminated and restarted), these changes are automatically reflected on the Dynamic Address Group, and Panorama ensures that these updates are immediately propagated to the management pods, which will consequently alter the configuration of firewall policies, ensuring a consistent security posture of resources at all times. This then allows sets of IP addresses automatically updated by the plugin to be used as the source and destination of a security policy, avoiding the need to personally monitor each change and manually manage the policy update each time.

The flexibility provided by these tags/labels allows us to achieve different levels of granularity when defining authorization policies, from DAGs at the node or cluster-level, to DAGs at the namespace and workload-level: for example, to isolate communication between pods within two different namespaces, using only namespace labels as source and destination DAGs. It is therefore clear that it is possible to specify and implement fine-grained policies, providing different types of checks and permissions performed by the firewall for each workload in the cluster.

Therefore, using Panorama, a different DAG was created for each workload within the cluster: one for each microservice, one for each instance of Postgres, one for the Kafka service and another for the Zookeeper service.

Figures 6.25 and 6.26 show the DAGs created for the Catalogue Service and its Postgres instance, respectively. Note that the type is 'dynamic' and that the matching criteria used to add IP addresses to these groups are different. In fact, for each DAG, the labels to be considered should be those relating to the various K8s resources associated with that workload, and thus the associated DaemonSet, ReplicaSet and clusterIP services exposed for that workload. These criteria were put into OR, since a workload at any given time can only be expressed as a source or destination using a single IP, which can be the actual IP of the pod or its virtual cluster IP. This ensures that traffic to that DAG service is allowed or denied whether the cluster IP or the pod IP is used as the destination of the packets: otherwise, since the clusterIP destination addresses are mapped to a corresponding pod IP by the host's kernel Iptables (Destination NAT), the firewall would not be able to match that destination address and could unexpectedly allow or deny traffic.

Figures 6.27 and 6.28, on the other hand, demonstrate how the K8s plugin actually retrieved the IP addresses associated with those specific K8s resources, namely the IP of the pod and the cluster IP of the Catalogue service and its postgres instance respectively, and placed them within the DAG associated with that service.

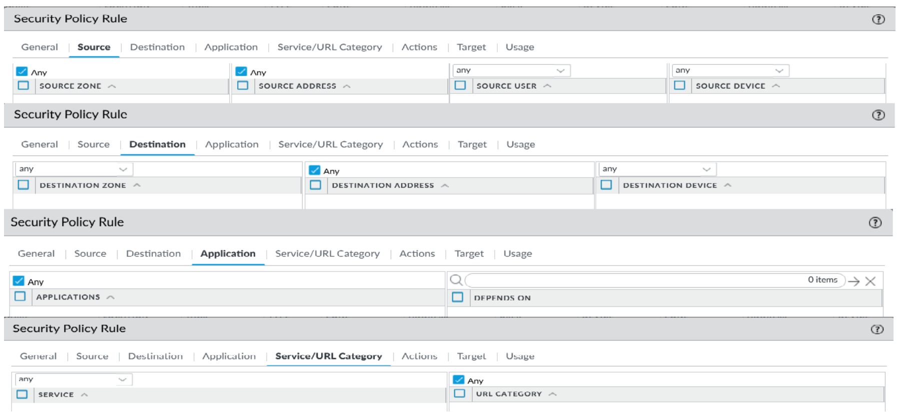
Once the DAGs to be used as sources and destinations for the security policies had been defined, the authorisation policies were created following the default deny approach, and thus explicitly permitting only the services expected between each pair of workloads, ensuring the *least privilege principle.*

First, therefore, a **Deny All** policy was created: as shown in Figure 6.29, this policy will match and deny any type of traffic, since the source and destination of the policy has been set to *any* zone, address, user and device, and the specific applications, services and URLs it impacts have also been set to *any.* This ensures that any traffic is denied by default unless an explicit authorisation policy has been defined for a specific tuple [source,destination,application,service]. The action was then set to deny (Figure 6.30), which implies that the corresponding packets will be dropped and no response will be returned to the source, generating a client-side timeout failure; however, other actions could have been specified, such as 'reset connection'. Finally, the rule has also been set up so that the firewall generates logs at the beginning and end of each session that corresponds to the denial policy: in this way, we will be able to inspect any anomalous traffic generated (or received) within the Kubernetes cluster, being able to monitor any security problems or attempts at lateral movement.

Since the packets are checked against firewall rules from top to bottom, and the first rule that matches the packet overrides the other rules below, the *Deny All* rule must be put at the bottom and all the other allow policies on top of it: in this way a deny policy will be applied only if no explicit allow policy matched that specific traffic.

I then defined explicit authorisation policies for each pair of services that must communicate, so that only that type of traffic is actually allowed. The following policies were then created:

- *An ALLOW policy for the DNS service*: since the various services running in the pods need access to the DNS service within the Kubernetes cluster for name resolution and to operate correctly, an explicit policy must be defined to allow such traffic that will otherwise be blocked by the deny all. This is why I defined two DAGs, one for the DNS service to be


Security Policy Rule

General | **Source** | Destination | Application | Service/URL Category | Actions | Target | Usage

- [✓] Any — SOURCE ZONE
- [✓] Any — SOURCE ADDRESS
- any — SOURCE USER
- any — SOURCE DEVICE

Security Policy Rule

General | Source | **Destination** | Application | Service/URL Category | Actions | Target | Usage

- any — DESTINATION ZONE
- [✓] Any — DESTINATION ADDRESS
- any — DESTINATION DEVICE

Security Policy Rule

General | Source | Destination | **Application** | Service/URL Category | Actions | Target | Usage

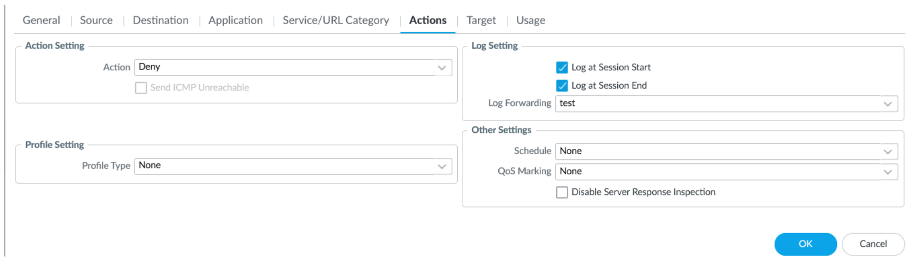
- [✓] Any — APPLICATIONS
- 0 items — DEPENDS ON

Security Policy Rule

General | Source | Destination | Application | **Service/URL Category** | Actions | Target | Usage

- any — SERVICE
- [✓] Any — URL CATEGORY

Figure 6.29: Deny All Policy Definition


General | Source | Destination | Application | Service/URL Category | **Actions** | Target | Usage

Action Setting
- Action: Deny
- [ ] Send ICMP Unreachable

Log Setting
- [✓] Log at Session Start
- [✓] Log at Session End
- Log Forwarding: test

Profile Setting
- Profile Type: None

Other Settings
- Schedule: None
- QoS Marking: None
- [ ] Disable Server Response Inspection

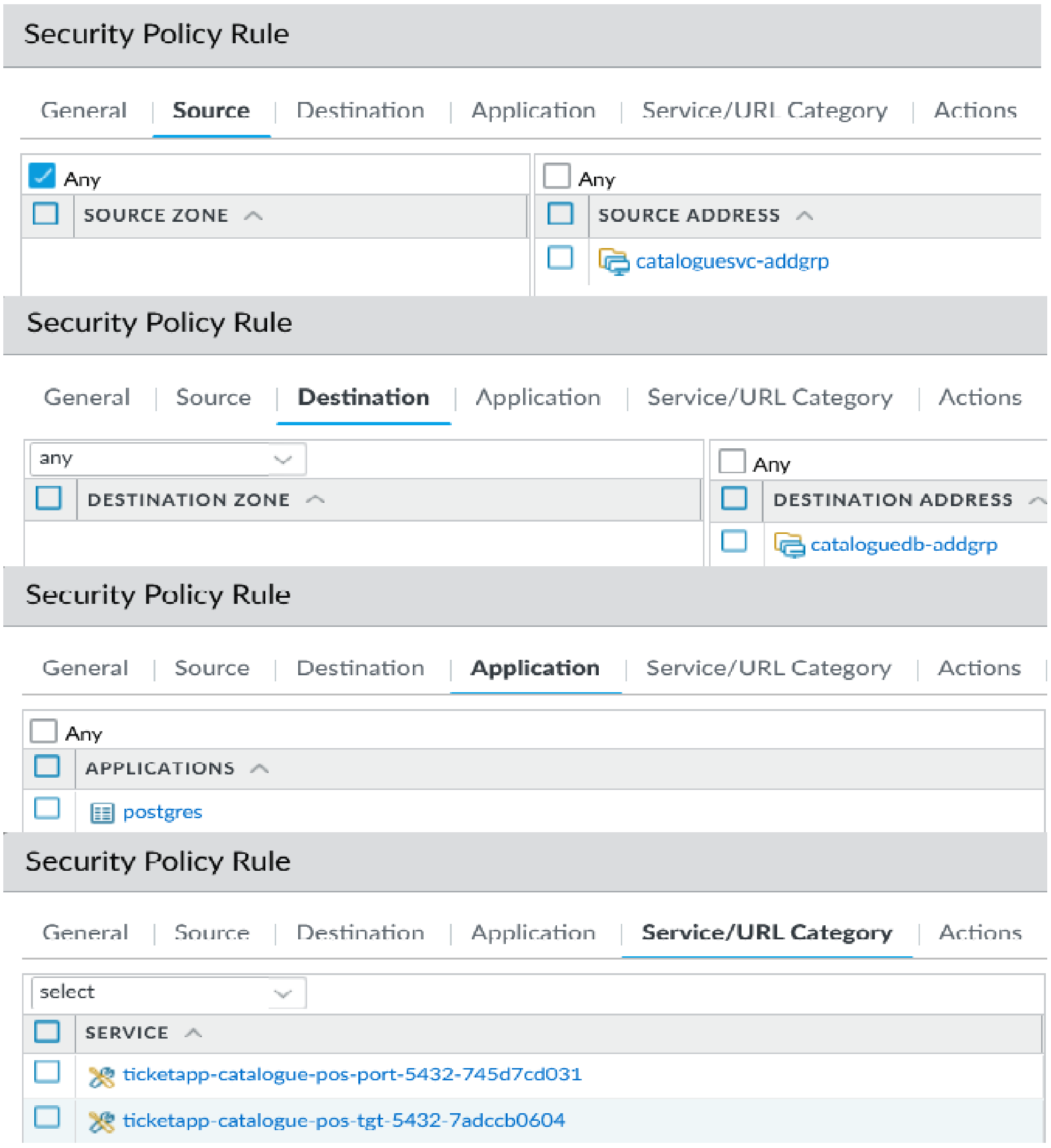
OK | Cancel

Figure 6.30: Deny All Policy Action and Logs Configuration

used as the destination, and another to allow all pods, services and nodes in the cluster to be included through a single explicit label.

- *An ALLOW policy for permitting web traffic between the Ingress Gateway and the four microservices (Login, Catalogue, Traveler and Payment)*: the gateway must be the only entity which can contact the web services exposed by the four workloads.
- *An ALLOW policy for permitting the four microservices (Login, Catalogue, Traveler and Payment) to contact their own specific Postgres instance*
- *An ALLOW policy for permitting only Catalogue, Traveler and Payment workloads to contact the Kafka service message broker*
- *An ALLOW policy for permitting the Kafka message broker to contact the Zookeeper service*

Figure 6.31 shows the *allow policy* created to allow the Catalogue Service pod to contact only its own instance of Postgres, allowing only Postgres application traffic on the specified port 5432 that is exposed by the service. Note that for each *rule allow*, the source and destination DAGs of the pods that need to communicate (i.e. the pods and IPs of the service) have been set, and also on which ports of the service they are allowed to communicate. In addition, the specific application type was also specified for each authorisation policy: since the CN-series firewall is in fact a Next Generation Firewall, it is able to inspect and allow/deny traffic not only at the OSI L3 and L4 levels, but also at the L7 level, achieving complete visibility of the content. Consequently, the CN-series is equipped with a series of application signatures that allow it to match and recognise


Security Policy Rule

General | **Source** | Destination | Application | Service/URL Category | Actions

✓ Any	Any
SOURCE ZONE	SOURCE ADDRESS
	cataloguesvc-addgrp

Security Policy Rule

General | Source | **Destination** | Application | Service/URL Category | Actions

any	Any
DESTINATION ZONE	DESTINATION ADDRESS
	cataloguedb-addgrp

Security Policy Rule

General | Source | Destination | **Application** | Service/URL Category | Actions

Any
APPLICATIONS
postgres

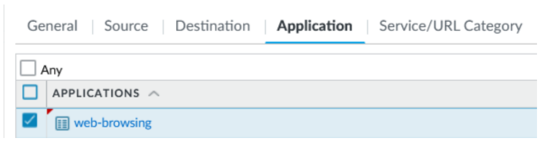
Security Policy Rule

General | Source | Destination | Application | **Service/URL Category** | Actions

select
SERVICE
ticketapp-catalogue-pos-port-5432-745d7cd031
ticketapp-catalogue-pos-tgt-5432-7adccb0604

Figure 6.31: Allow 'Catalogue to DB' Policy Definition

the type of traffic exchanged regardless of the port used. It is also possible to create a signature for your own customised application, in case it differs from the known standard ones. Therefore, by also specifying the type of application traffic, we are able to provide stronger authorisation measures. The Figures 6.32, 6.33, 6.34, and 6.35 show the specific application types which the CN-series is able to recognize and that I set for the various allow policies (i.e. the Postgres, Kafka, Zookeper and Web traffic type).

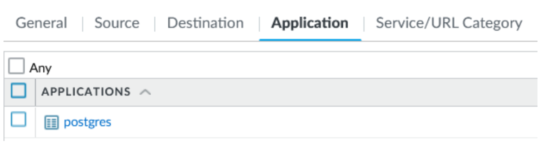

General | Source | Destination | **Application** | Service/URL Category

☐ Any

☐ APPLICATIONS

☑ web-browsing

Figure 6.32: Policy For Web-Browsing Application Traffic

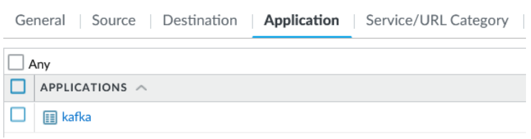

General | Source | Destination | **Application** | Service/URL Category

☐ Any

☐ APPLICATIONS

☐ postgres

Figure 6.33: Policy For Postgres Application Traffic

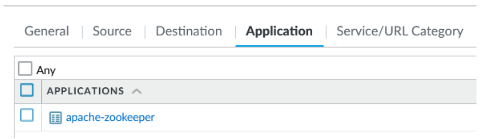

General | Source | Destination | **Application** | Service/URL Category

☐ Any

☐ APPLICATIONS

☐ kafka

Figure 6.34: Policy For Kafka Application Traffic

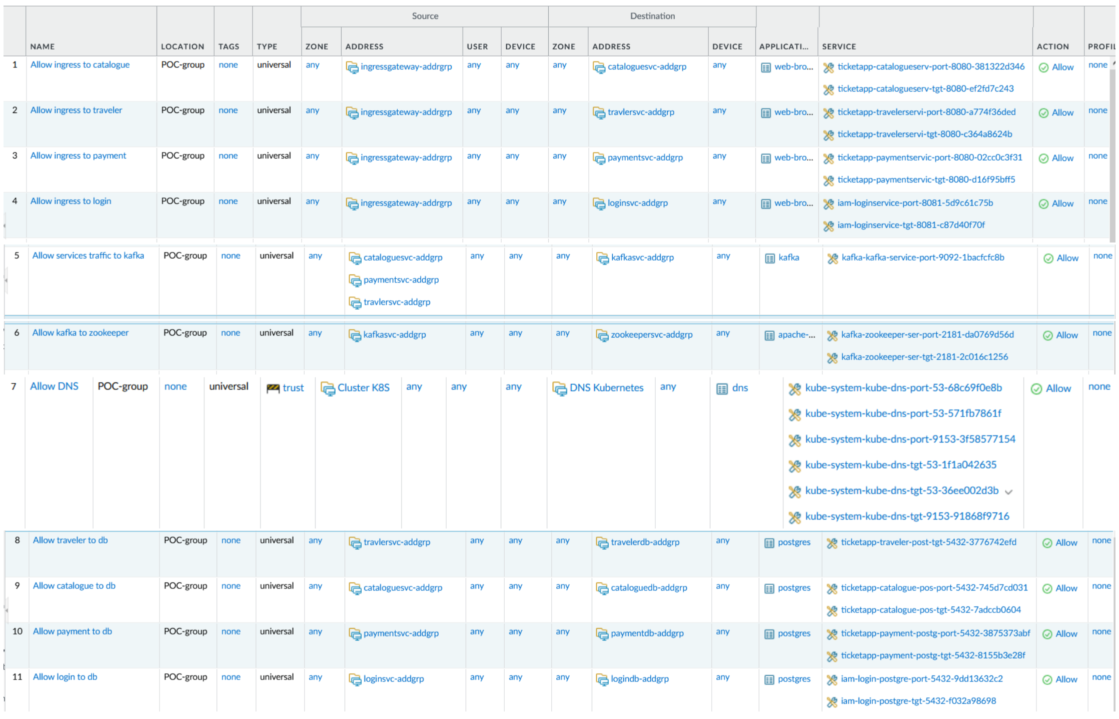

General | Source | Destination | **Application** | Service/URL Category

☐ Any

☐ APPLICATIONS

☐ apache-zookeeper

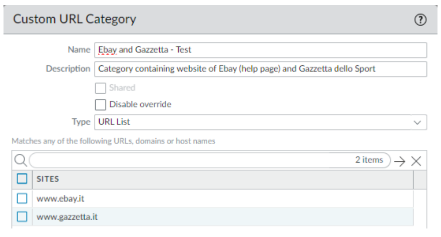
Figure 6.35: Policy For Zookeeper Application Traffic



					Source				Destination						
	NAME	LOCATION	TAGS	TYPE	ZONE	ADDRESS	USER	DEVICE	ZONE	ADDRESS	DEVICE	APPLICATI...	SERVICE	ACTION	PROFIL
1	Allow ingress to catalogue	POC-group	none	universal	any	ingressgateway-addrgrp	any	any	any	cataloguesvc-addgrp	any	web-bro...	ticketapp-catalogueserv-port-8080-381322d346 ticketapp-catalogueserv-tgt-8080-ef2fd7c243	Allow	none
2	Allow ingress to traveler	POC-group	none	universal	any	ingressgateway-addrgrp	any	any	any	travlersvc-addgrp	any	web-bro...	ticketapp-travelerservi-port-8080-a774f36ded ticketapp-travelerservi-tgt-8080-c364a8624b	Allow	none
3	Allow ingress to payment	POC-group	none	universal	any	ingressgateway-addrgrp	any	any	any	paymentsvc-addgrp	any	web-bro...	ticketapp-paymentservic-port-8080-02cc0c3f31 ticketapp-paymentservic-tgt-8080-d16f95bff5	Allow	none
4	Allow ingress to login	POC-group	none	universal	any	ingressgateway-addrgrp	any	any	any	loginsvc-addgrp	any	web-bro...	iam-loginservice-port-8081-5d9c61c75b iam-loginservice-tgt-8081-c87d40f70f	Allow	none
5	Allow services traffic to kafka	POC-group	none	universal	any	cataloguesvc-addgrp paymentsvc-addgrp travlersvc-addgrp	any	any	any	kafkasvc-addgrp	any	kafka	kafka-kafka-service-port-9092-1bacfcfc8b	Allow	none
6	Allow kafka to zookeeper	POC-group	none	universal	any	kafkasvc-addgrp	any	any	any	zookeepersvc-addgrp	any	apache-...	kafka-zookeeper-ser-port-2181-da0769d56d kafka-zookeeper-ser-tgt-2181-2c016c1256	Allow	none
7	Allow DNS	POC-group	none	universal	trust	Cluster K8S	any	any	any	DNS Kubernetes	any	dns	kube-system-kube-dns-port-53-68c69f0e8b kube-system-kube-dns-port-53-571fb7861f kube-system-kube-dns-port-9153-3f58577154 kube-system-kube-dns-tgt-53-1f1a042635 kube-system-kube-dns-tgt-53-36ee002d3b kube-system-kube-dns-tgt-9153-91868f9716	Allow	none
8	Allow traveler to db	POC-group	none	universal	any	travlersvc-addgrp	any	any	any	travelerdb-addgrp	any	postgres	ticketapp-traveler-post-tgt-5432-3776742efd	Allow	none
9	Allow catalogue to db	POC-group	none	universal	any	cataloguesvc-addgrp	any	any	any	cataloguedb-addgrp	any	postgres	ticketapp-catalogue-pos-port-5432-745d7cd031 ticketapp-catalogue-pos-tgt-5432-7adccb0604	Allow	none
10	Allow payment to db	POC-group	none	universal	any	paymentsvc-addgrp	any	any	any	paymentdb-addgrp	any	postgres	ticketapp-payment-postg-port-5432-3875373abf ticketapp-payment-postg-tgt-5432-8155b3e28f	Allow	none
11	Allow login to db	POC-group	none	universal	any	loginsvc-addgrp	any	any	any	logindb-addgrp	any	postgres	iam-login-postgre-port-5432-9dd13632c2 iam-login-postgre-tgt-5432-f032a98698	Allow	none

Figure 6.36: Overview of the Allow Policies created

The Figure 6.36 shows an overview of the created ALLOW policies described above.



Custom URL Category

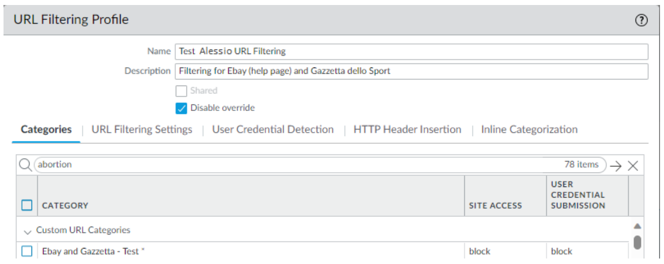
Name	Ebay and Gazzetta - Test
Description	Category containing website of Ebay (help page) and Gazzetta dello Sport
	☐ Shared
	☐ Disable override
Type	URL List

Matches any of the following URLs, domains or host names

2 items

SITES
www.ebay.it
www.gazzetta.it

Figure 6.37: URL Category



URL Filtering Profile

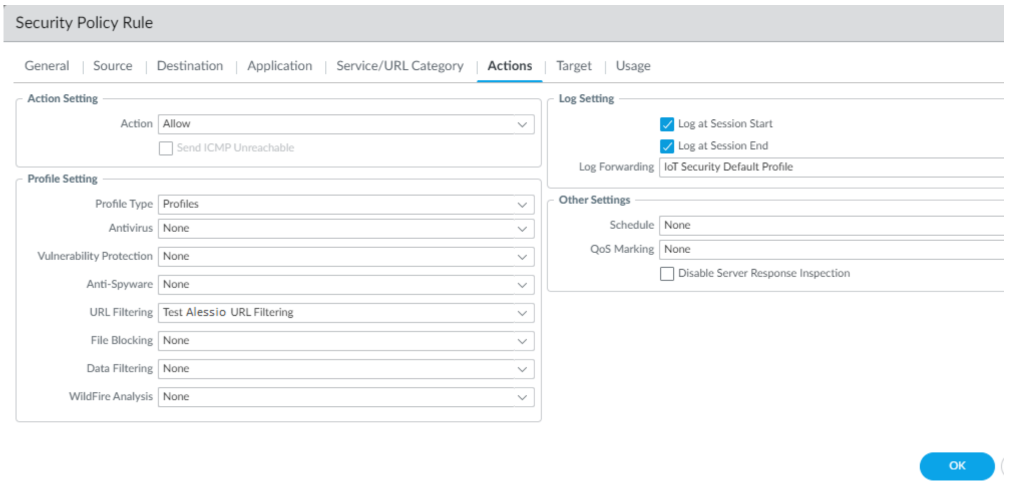
Name	Test Alessio URL Filtering
Description	Filtering for Ebay (help page) and Gazzetta dello Sport
	☐ Shared
	✓ Disable override

Categories | URL Filtering Settings | User Credential Detection | HTTP Header Insertion | Inline Categorization

abortion — 78 items

CATEGORY	SITE ACCESS	USER CREDENTIAL SUBMISSION
Custom URL Categories		
Ebay and Gazzetta - Test *	block	block

Figure 6.38: URL Filtering Profile



Security Policy Rule

General | Source | Destination | Application | Service/URL Category | Actions | Target | Usage

Action Setting

Action	Allow
	☐ Send ICMP Unreachable

Profile Setting

Profile Type	Profiles
Antivirus	None
Vulnerability Protection	None
Anti-Spyware	None
URL Filtering	Test Alessio URL Filtering
File Blocking	None
Data Filtering	None
WildFire Analysis	None

Log Setting

	✓ Log at Session Start
	✓ Log at Session End
Log Forwarding	IoT Security Default Profile

Other Settings

Schedule	None
QoS Marking	None
	☐ Disable Server Response Inspection

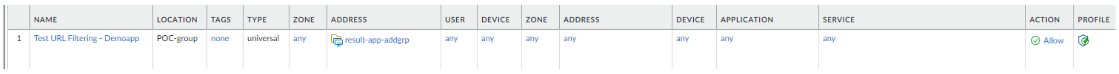
Figure 6.39: URL Filtering Policy Definition

6.2.3 Deep Packet Inspection

In order to test the deep packet inspection features of the Next Generation Firewall, the *URL Filtering*and the *Advanced Threat Prevention* capabilities were considered. Therefore two specific security policies were defined.

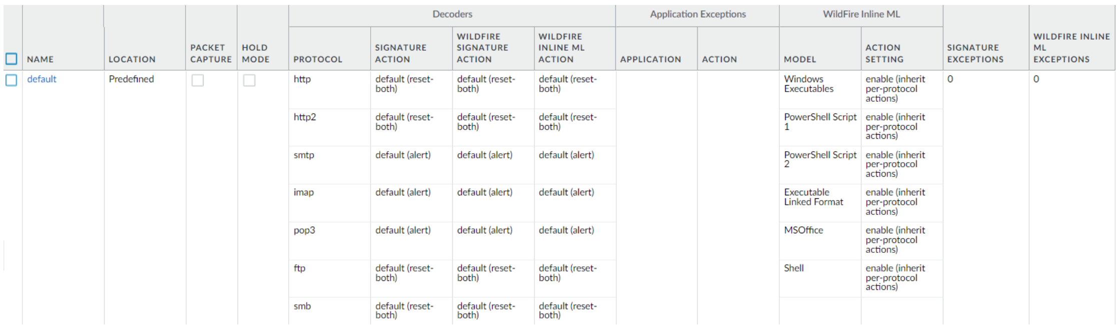
For the URL filtering policy, it was first necessary to create a customised URL category with some URLs associated with it, then, as shown in Figure 6.37, a trivial category including the URLs *www.ebay.it* and *www.gazzetta.it.* was created. It was then necessary to create a URL filtering profile and add the previously created category (Figure 6.38) to it. This is necessary to specify the URLs that you wish to filter. Finally, a security policy has been created that allows outbound traffic to any destination, but implements the URL filtering function with respect to the previously defined URL filtering profile (Figure 6.39). This will allow the specified source to contact any destination, but any outgoing connection to these URLs will be prevented by the packet inspection performed by the firewall at L7. This is extremely useful for setting a category of malicious website or repository URLs and preventing outgoing traffic regardless of source. Figure 6.40 shows instead the final policy.

As far as the Advanced Threat Prevention functionality is concerned, I decided to test the antivirus capability. This involved first of all defining an antivirus security profile (a default



	NAME	LOCATION	TAGS	TYPE	ZONE	ADDRESS	USER	DEVICE	ZONE	ADDRESS	DEVICE	APPLICATION	SERVICE	ACTION	PROFILE
1	Test URL Filtering - Demoapp	POC-group	none	universal	any	result-app-addgrp	any	any	any	any	any	any	any	✓ Allow	

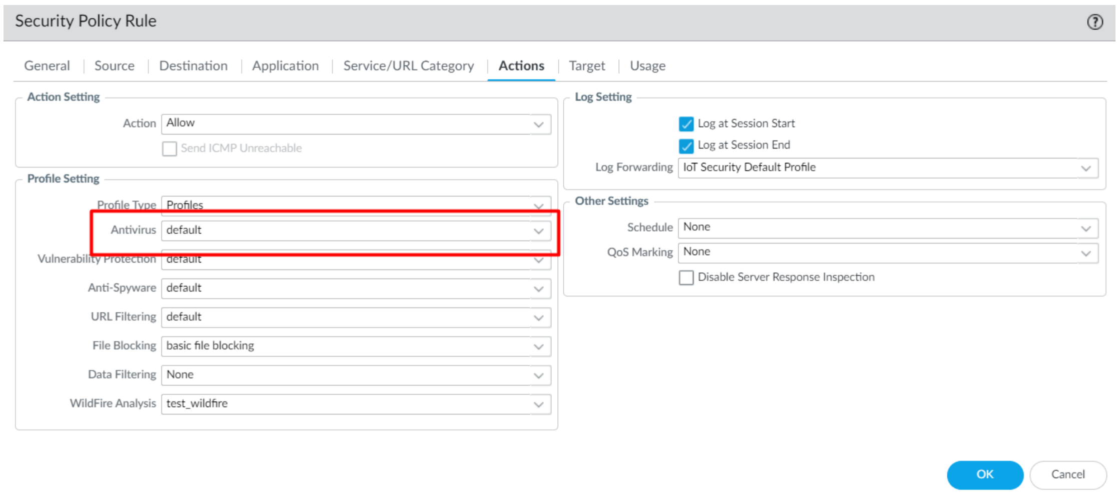
Figure 6.40: URL Filtering Policy



					Decoders				Application Exceptions		WildFire Inline ML			
☐	NAME	LOCATION	PACKET CAPTURE	HOLD MODE	PROTOCOL	SIGNATURE ACTION	WILDFIRE SIGNATURE ACTION	WILDFIRE INLINE ML ACTION	APPLICATION	ACTION	MODEL	ACTION SETTING	SIGNATURE EXCEPTIONS	WILDFIRE INLINE ML EXCEPTIONS
☐	default	Predefined	☐	☐	http	default (reset-both)	default (reset-both)	default (reset-both)			Windows Executables	enable (inherit per-protocol actions)	0	0
					http2	default (reset-both)	default (reset-both)	default (reset-both)			PowerShell Script 1	enable (inherit per-protocol actions)		
					smtp	default (alert)	default (alert)	default (alert)			PowerShell Script 2	enable (inherit per-protocol actions)		
					imap	default (alert)	default (alert)	default (alert)			Executable Linked Format	enable (inherit per-protocol actions)		
					pop3	default (alert)	default (alert)	default (alert)			MSOffice	enable (inherit per-protocol actions)		
					ftp	default (reset-both)	default (reset-both)	default (reset-both)			Shell	enable (inherit per-protocol actions)		
					smb	default (reset-both)	default (reset-both)	default (reset-both)						

Figure 6.41: Antivirus Policy Profile

profile was already provided, as shown in Figure 6.41), and then defining a security policy and setting within the configuration the profile to be adopted during packet inspection (Figure 6.42). The CN-series is equipped with a database of virus signatures, and by leveraging these signatures, which are always up-to-date, it is able to recognise traces of viruses in the traffic exchanged.


Security Policy Rule

General | Source | Destination | Application | Service/URL Category | **Actions** | Target | Usage

Action Setting

Action	Allow
☐	Send ICMP Unreachable

Profile Setting

Profile Type	Profiles
Antivirus	default
Vulnerability Protection	default
Anti-Spyware	default
URL Filtering	default
File Blocking	basic file blocking
Data Filtering	None
WildFire Analysis	test_wildfire

Log Setting

✓	Log at Session Start
✓	Log at Session End
Log Forwarding	IoT Security Default Profile

Other Settings

Schedule	None
QoS Marking	None
☐	Disable Server Response Inspection

Figure 6.42: Antivirus Policy Configuration

Chapter 7

Results And Proof Of Concept Validation

7.1 Istio

7.1.1 mTLS, Peer Authentication And Authorization

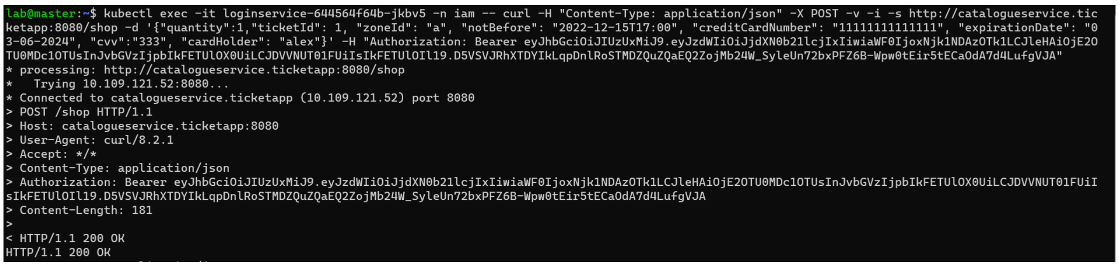
To test whether communication between two workloads is actually updated to an mTLS communication, I used *curl*, a command-line tool that developers use to transfer data to and from a server. It was specially installed inside each container image produced for our microservices to test this type of functionality. Then I adopted wireshark, a network packet sniffer that is very popular nowadays, so that I could listen in on the virtual network interface of the pod server, and thus be able to intercept and see all the traffic received, inspecting its contents. Since the containers in the same pod share the same network namespace and virtual network interface (and thus IP address), the traffic received on the pod's virtual interface from outside will actually be the traffic destined for the sidecar proxy, which will then be passed internally to the main container. This means that we expected the traffic received by the pod to be TLS.

Therefore, I moved to use the *kubectl exec* command to execute curl inside the Login service pod, deployed in the iam namespace, and send a request to the Catalogue service, deployed instead in the ticketapp namespace. Before sending the request, I logged in the application through the login service and retrieved a valid JWT. After that, I decided to make the login service (whose pod has an injected istio proxy) contact the POST /shop endpoint of the Catalogue Service, providing a valid JSON as the body of my request. This way I could easily verify whether my request will be encrypted after sending or not. The JWT was placed within the request as well, since /shop is an authenticated endpoint. Figure 7.1 shows the request sent through curl by the login service in the iam namespace, and the answer received: the request was accepted and a *200 - OK* message was returned, and thus the communication was necessarily successful. Figure 7.2 instead, shows what I saw on the Catalogue Service pod network interface with wireshark: it is clear from the picture that a TLS connection comes from the IP address of the login service pod, and after exchanging the TLS handshake messages to authenticate each other and negotiate cryptographic parameters, application data are exchanged over the established secure channel. In fact, by inspecting the content of these data, it is possible to see that they are actually encrypted (Figure 7.3).

But neither the Login service nor the Catalogue service were configured with TLS settings, so this necessarily meant that the client's proxy automatically upgraded outgoing plaintext traffic to mTLS, whereas since the connection received sniffed on the server's pod was TLS but the Catalogue application container within the pod does not accept TLS traffic, the server's proxy necessarily acted as a TLS tunnel terminator, decrypting the traffic and passing it on to the workload on localhost interface.

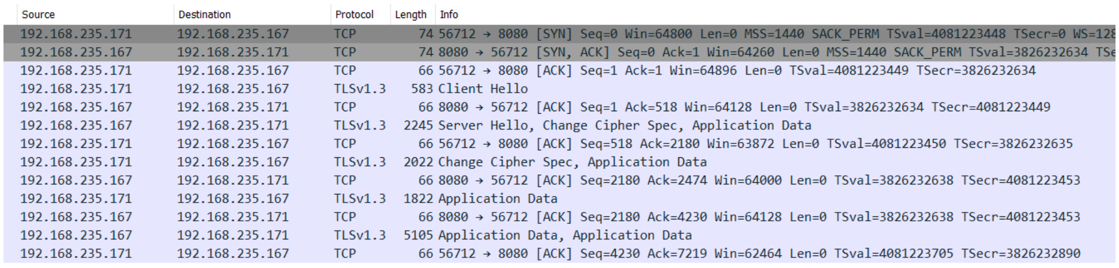
Next, I went on to test whether, indeed, the PeerAuthentication policy set in STRICT mode prevented proxies operating as servers from accepting plaintext traffic (not mTLS). To do this, I created a temporary namespace called *test-nomtls*, and deployed the login service in it: in this way, I got the same client service, but without any proxy histio injected, since I did not label the namespace test-nomtls. I then repeated a similar test as before, but this time the outgoing traffic from the login was plaintext (since no proxy was upgrading it to mTLS traffic). By listening on the same virtual interface of the Catalogue service pod, and sending the same request as before, I could actually see that the traffic received was plain TCP traffic, and above all I could actually confirm that the connection, not being TLS (and thus neither authenticated nor protected), was automatically dropped by the proxy server, and no reply was provided. (See Figures 7.4 and 7.5)

Please note, however, that these two tests were carried out shortly before setting and applying any authorisation policy, otherwise traffic between the two workloads login service and catalogue service would not have been possible, since they are not supposed to communicate directly with each other.



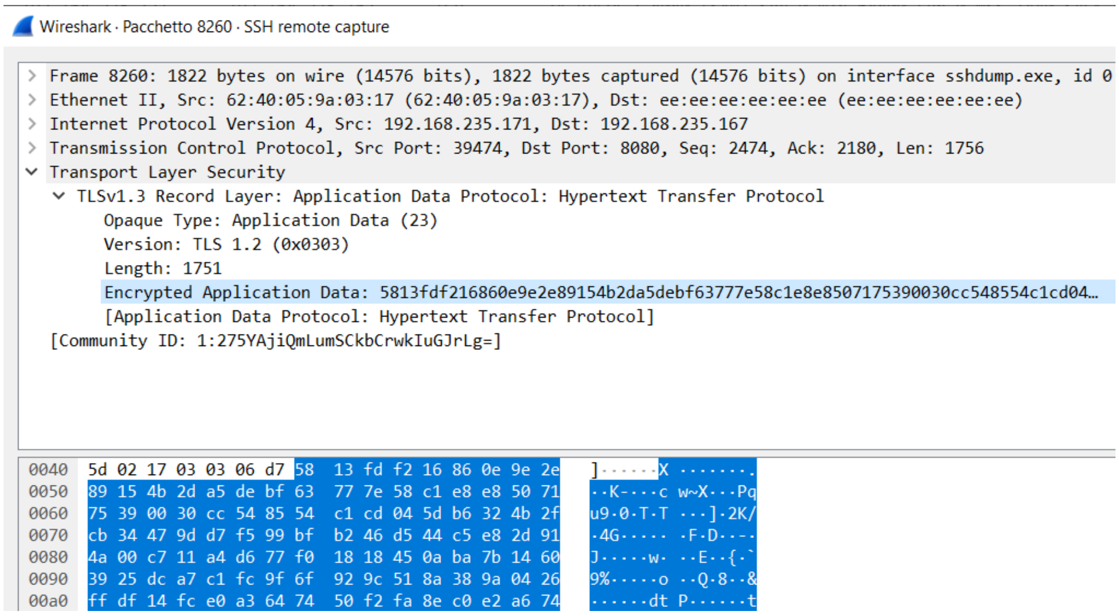
```
lab@master:~$ kubectl exec -it loginservice-644564f64b-jkbv5 -n iam -- curl -H "Content-Type: application/json" -X POST -v -i -s http://catalogueservice.tic
ketapp:8080/shop -d '{"quantity":1,"ticketId": 1, "zoneId": "a", "notBefore": "2022-12-15T17:00", "creditCardNumber": "11111111111111", "expirationDate": "0
3-06-2024", "cvv":"333", "cardHolder": "alex"}' -H "Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwiaWF0IjoxNjk1NDAzOTk1LCJleHAiOjE2O
TU0MDc1OTUsInJvbGVzIjpbIkFETUlOX0UiLCJDVVNUT01FUiIsIkFETUlOIl19.D5VSVJRhXTDYIkLqpDnlRoSTMDZQuZQaEQ2ZojMb24W_SyleUn72bxPFZ6B-Wpw0tEir5tECaOdA7d4LufgVJA"
* processing: http://catalogueservice.ticketapp:8080/shop
*   Trying 10.109.121.52:8080...
* Connected to catalogueservice.ticketapp (10.109.121.52) port 8080
> POST /shop HTTP/1.1
> Host: catalogueservice.ticketapp:8080
> User-Agent: curl/8.2.1
> Accept: */*
> Content-Type: application/json
> Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwiaWF0IjoxNjk1NDAzOTk1LCJleHAiOjE2OTU0MDc1OTUsInJvbGVzIjpbIkFETUlOX0UiLCJDVVNUT01FUiI
sIkFETUlOIl19.D5VSVJRhXTDYIkLqpDnlRoSTMDZQuZQaEQ2ZojMb24W_SyleUn72bxPFZ6B-Wpw0tEir5tECaOdA7d4LufgVJA
> Content-Length: 181
>
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
```

Figure 7.1: Test Peer Authentication



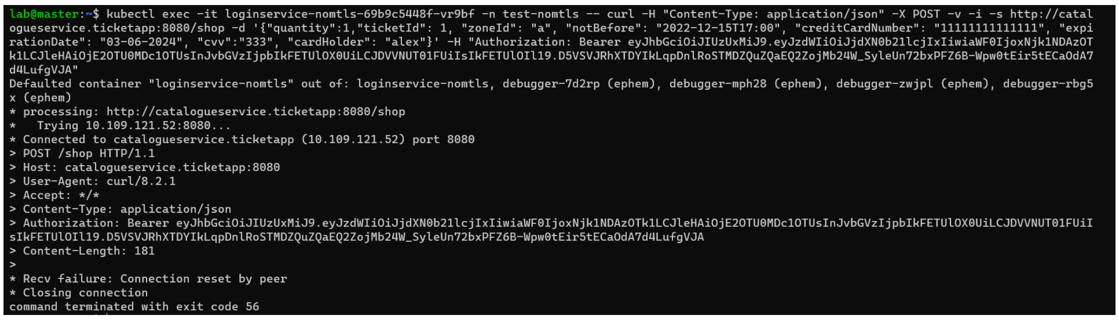
```
 Source            Destination       Protocol  Length Info
 192.168.235.171   192.168.235.167   TCP           74 56712 → 8080 [SYN] Seq=0 Win=64800 Len=0 MSS=1440 SACK_PERM TSval=4081223448 TSecr=0 WS=128
 192.168.235.167   192.168.235.171   TCP           74 8080 → 56712 [SYN, ACK] Seq=0 Ack=1 Win=64260 Len=0 MSS=1440 SACK_PERM TSval=3826232634 TSe
 192.168.235.171   192.168.235.167   TCP           66 56712 → 8080 [ACK] Seq=1 Ack=1 Win=64896 Len=0 TSval=4081223449 TSecr=3826232634
 192.168.235.171   192.168.235.167   TLSv1.3      583 Client Hello
 192.168.235.167   192.168.235.171   TCP           66 8080 → 56712 [ACK] Seq=1 Ack=518 Win=64128 Len=0 TSval=3826232634 TSecr=4081223449
 192.168.235.167   192.168.235.171   TLSv1.3     2245 Server Hello, Change Cipher Spec, Application Data
 192.168.235.171   192.168.235.167   TCP           66 56712 → 8080 [ACK] Seq=518 Ack=2180 Win=63872 Len=0 TSval=4081223450 TSecr=3826232635
 192.168.235.171   192.168.235.167   TLSv1.3     2022 Change Cipher Spec, Application Data
 192.168.235.167   192.168.235.171   TCP           66 8080 → 56712 [ACK] Seq=2180 Ack=2474 Win=64000 Len=0 TSval=3826232638 TSecr=4081223453
 192.168.235.171   192.168.235.167   TLSv1.3     1822 Application Data
 192.168.235.167   192.168.235.171   TCP           66 8080 → 56712 [ACK] Seq=2180 Ack=4230 Win=64128 Len=0 TSval=3826232638 TSecr=4081223453
 192.168.235.167   192.168.235.171   TLSv1.3     5105 Application Data, Application Data
 192.168.235.171   192.168.235.167   TCP           66 56712 → 8080 [ACK] Seq=4230 Ack=7219 Win=62464 Len=0 TSval=4081223705 TSecr=3826232890
```

Figure 7.2: TLS traffic exchanged



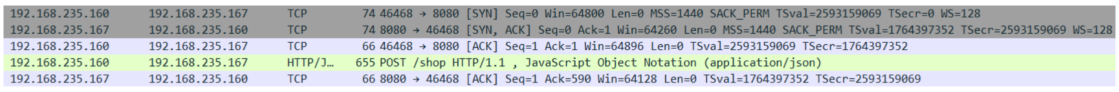
```
Wireshark · Pacchetto 8260 · SSH remote capture

> Frame 8260: 1822 bytes on wire (14576 bits), 1822 bytes captured (14576 bits) on interface sshdump.exe, id 0
> Ethernet II, Src: 62:40:05:9a:03:17 (62:40:05:9a:03:17), Dst: ee:ee:ee:ee:ee:ee (ee:ee:ee:ee:ee:ee)
> Internet Protocol Version 4, Src: 192.168.235.171, Dst: 192.168.235.167
> Transmission Control Protocol, Src Port: 39474, Dst Port: 8080, Seq: 2474, Ack: 2180, Len: 1756
v Transport Layer Security
  v TLSv1.3 Record Layer: Application Data Protocol: Hypertext Transfer Protocol
      Opaque Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 1751
      Encrypted Application Data: 5813fdf216860e9e2e89154b2da5debf63777e58c1e8e8507175390030cc548554c1cd04…
      [Application Data Protocol: Hypertext Transfer Protocol]
  [Community ID: 1:275YAjiQmLumSCkbCrwkIuGJrLg=]

0040  5d 02 17 03 03 06 d7 58  13 fd f2 16 86 0e 9e 2e   ]······X ········.
0050  89 15 4b 2d a5 de bf 63  77 7e 58 c1 e8 e8 50 71   ··K-···c w~X···Pq
0060  75 39 00 30 cc 54 85 54  c1 cd 04 5d b6 32 4b 2f   u9·0·T·T ···]·2K/
0070  cb 34 47 9d d7 f5 99 bf  b2 46 d5 44 c5 e8 2d 91   ·4G····· ·F·D··-·
0080  4a 00 c7 11 a4 d6 77 f0  18 18 45 0a ba 7b 14 60   J·····w· ··E··{·`
0090  39 25 dc a7 c1 fc 9f 6f  92 9c 51 8a 38 9a 04 26   9%·····o ··Q·8··&
00a0  ff df 14 fc e0 a3 64 74  50 f2 fa 8e c0 e2 a6 74   ······dt P······t
```

Figure 7.3: TLS traffic content



```
lab@master:~$ kubectl exec -it loginservice-nomtls-69b9c5448f-vr9bf -n test-nomtls -- curl -H "Content-Type: application/json" -X POST -v -i -s http://catal
ogueservice.ticketapp:8080/shop -d '{"quantity":1,"ticketId": 1, "zoneId": "a", "notBefore": "2022-12-15T17:00", "creditCardNumber": "11111111111111", "expi
rationDate": "03-06-2024", "cvv":"333", "cardHolder": "alex"}' -H "Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwiaWF0IjoxNjk1NDAzOT
k1LCJleHAiOjE2OTU0MDc1OTUsInJvbGVzIjpbIkFETUlOX0UiLCJDVVNUT01FUiIsIkFETUlOIl19.D5VSVJRhXTDYIkLqpDnlRoSTMDZQuZQaEQ2ZojMb24W_SyleUn72bxPFZ6B-Wpw0tEir5tECaOdA7
d4LufgVJA"
Defaulted container "loginservice-nomtls" out of: loginservice-nomtls, debugger-7d2rp (ephem), debugger-mph28 (ephem), debugger-zwjpl (ephem), debugger-rbg5
x (ephem)
* processing: http://catalogueservice.ticketapp:8080/shop
*   Trying 10.109.121.52:8080...
* Connected to catalogueservice.ticketapp (10.109.121.52) port 8080
> POST /shop HTTP/1.1
> Host: catalogueservice.ticketapp:8080
> User-Agent: curl/8.2.1
> Accept: */*
> Content-Type: application/json
> Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwiaWF0IjoxNjk1NDAzOTk1LCJleHAiOjE2OTU0MDc1OTUsInJvbGVzIjpbIkFETUlOX0UiLCJDVVNUT01FUiI
sIkFETUlOIl19.D5VSVJRhXTDYIkLqpDnlRoSTMDZQuZQaEQ2ZojMb24W_SyleUn72bxPFZ6B-Wpw0tEir5tECaOdA7d4LufgVJA
> Content-Length: 181
>
* Recv failure: Connection reset by peer
* Closing connection
command terminated with exit code 56
```

Figure 7.4: Test without mTLS

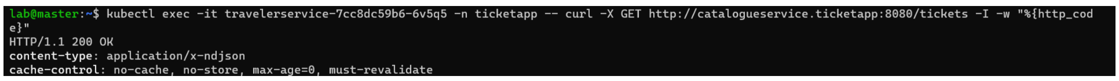


```
192.168.235.160   192.168.235.167   TCP      74 46468 → 8080 [SYN] Seq=0 Win=64800 Len=0 MSS=1440 SACK_PERM TSval=2593159069 TSecr=0 WS=128
192.168.235.167   192.168.235.160   TCP      74 8080 → 46468 [SYN, ACK] Seq=0 Ack=1 Win=64260 Len=0 MSS=1440 SACK_PERM TSval=1764397352 TSecr=2593159069 WS=128
192.168.235.160   192.168.235.167   TCP      66 46468 → 8080 [ACK] Seq=1 Ack=1 Win=64896 Len=0 TSval=2593159069 TSecr=1764397352
192.168.235.160   192.168.235.167   HTTP/J…  655 POST /shop HTTP/1.1 , JavaScript Object Notation (application/json)
192.168.235.167   192.168.235.160   TCP      66 8080 → 46468 [ACK] Seq=1 Ack=590 Win=64128 Len=0 TSval=1764397352 TSecr=2593159069
```

Figure 7.5: Non-mTLS traffic content

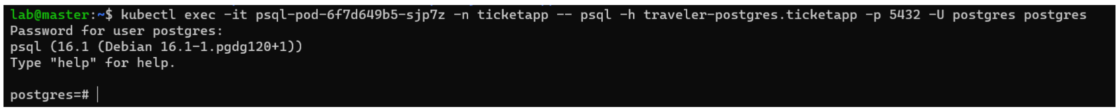
After enforcing the AuthenticationPolicy in STRICT mode on our workloads, only mTLS traffic (and therefore authenticated and protected) is accepted. However, before applying the AuthorizationPolicy seen in section 6.1, there is no restriction on incoming mTLS authenticated traffic: so at this very moment, with no authorisation policy applied, any pod within the network is able to contact our services/pods, provided it uses a valid X.509 certificate and authenticates itself correctly. Just before deploying the Istio authorization policies thus, some tests were carried out in order to demonstrate it: as you can see from the test performed in Figure 7.6 for example, the Traveler service is able to contact the Catalogue service endpoints without any problem, receiving a valid response (*HTTP/1.1 200 OK*) although this was not considered a normal behaviour for the Traveler service. In fact, the Catalogue Service (as well as any other microservice) should only be allowed to be contacted by the Ingress Gateway on the specific exposed port 8080.

Then, to demonstrate that any other service is also able to contact any instance of the Postgres database (which instead should only be allowed to the specific microservice, as it stores its own private data), a service *psql* was deployed within a pod in the ticketapp namespace: psql (PostgreSQL) is a terminal-based front-end for PostgreSQL, which allows one to connect to the DBMS and type queries interactively, send them to PostgreSQL and see the results. The pod mounted a different but valid service account, and since it was deployed within the ticketapp namespace, an Envoy proxy with a valid certificate was automatically injected into it, and thus was able to produce valid mTLS traffic. In Figure 7.7 it is shown how, by simply running psql and specifying the hostname, port and username, we connected to the Postgres DBMS and started the authentication process (since each instance of Postgres is protected by a password) and, once the correct password was provided, we gained access to the contents of the Traveler Service database and could easily exfiltrate any sensitive data. The communication was immediately accepted as it was correctly authenticated with mTLS and protected, and no authorisation policy was applied.



```
lab@master:~$ kubectl exec -it travelerservice-7cc8dc59b6-6v5q5 -n ticketapp -- curl -X GET http://catalogueservice.ticketapp:8080/tickets -I -w "%{http_cod
e}"
HTTP/1.1 200 OK
content-type: application/x-ndjson
cache-control: no-cache, no-store, max-age=0, must-revalidate
```

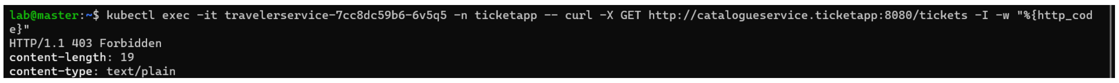
Figure 7.6: Test 1 before authorization policy



```
lab@master:~$ kubectl exec -it psql-pod-6f7d649b5-sjp7z -n ticketapp -- psql -h traveler-postgres.ticketapp -p 5432 -U postgres postgres
Password for user postgres:
psql (16.1 (Debian 16.1-1.pgdg120+1))
Type "help" for help.

postgres=# |
```

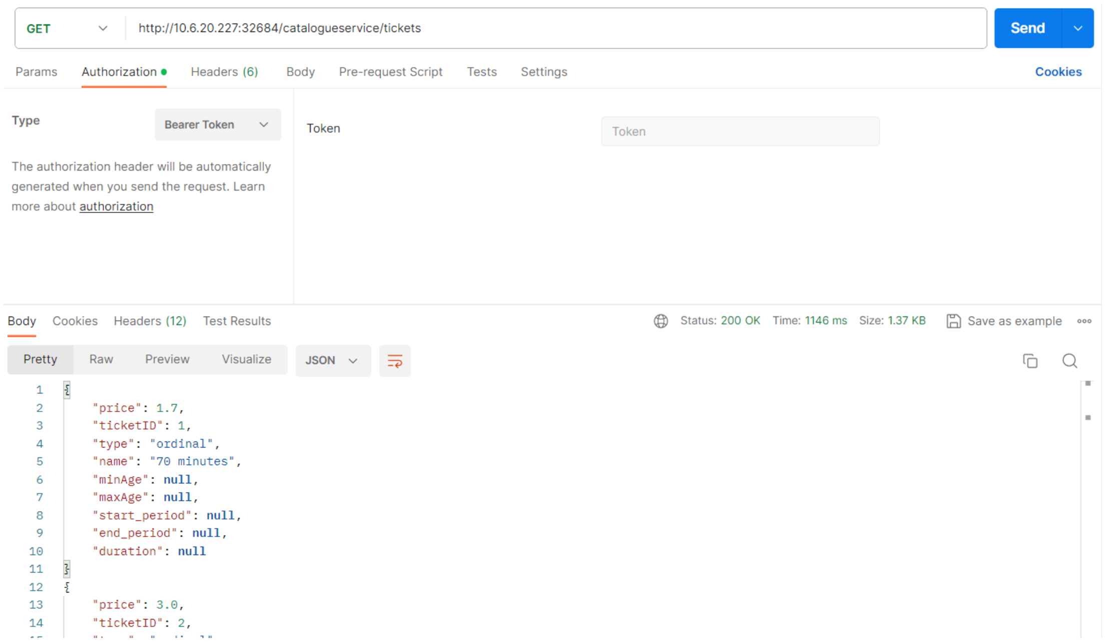
Figure 7.7: Test 2 before authorization policy

Subsequently, after having applied the AuthorizationPolicy instead, we repeated the same tests to see whether they actually worked as intended. Figure 7.8 shows how this time, instead of getting the *200 OK* response, a *HTTP/1.1 403 Forbidden* was returned: this is the HTTP response automatically generated by the proxy whenever an authorisation policy is applied and the connection or the request is denied. We then tried immediately afterwards to contact the same service via the Ingress Gateway, to verify that, indeed the latter is the only one capable of contacting the service and providing its functionality by acting as an intermediary. This time a platform for API testing called Postman was used from my own personal computer, allowing me to easily send a GET request to the same Catalogue service endpoint from outside: the test performed, shown in Figure 7.9, demonstrated that indeed traffic is allowed in this case, because if this had not been the case, contacting the Ingress gateway would not have resulted in any valid response. Instead, a *200 OK* response was returned, along with a list of tickets in JSON format within its body. This not only proved that ALLOW authorization policies work well on the Catalogue service, but also on the Postgres instance of the Catalogue, since without access to its database, the microservice would not have been able to retrieve and send the list of tickets. Finally, Figure 7.10 shows how now, the same psql service can no longer contact the traveler's database: the connection was closed by the server due to the Authorization policy, and now only the Traveler service can actually access it now.



```
lab@master:~$ kubectl exec -it travelerservice-7cc8dc59b6-6v5q5 -n ticketapp -- curl -X GET http://catalogueservice.ticketapp:8080/tickets -I -w "%{http_cod
e}"
HTTP/1.1 403 Forbidden
content-length: 19
content-type: text/plain
```

Figure 7.8: Test 3



GET http://10.6.20.227:32684/catalogueservice/tickets Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

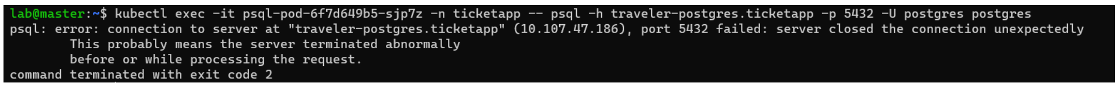
Token: Token

Body Cookies Headers (12) Test Results — Status: 200 OK Time: 1146 ms Size: 1.37 KB Save as example

Pretty Raw Preview Visualize JSON

```
{
    "price": 1.7,
    "ticketID": 1,
    "type": "ordinal",
    "name": "70 minutes",
    "minAge": null,
    "maxAge": null,
    "start_period": null,
    "end_period": null,
    "duration": null
}
{
    "price": 3.0,
    "ticketID": 2,
```

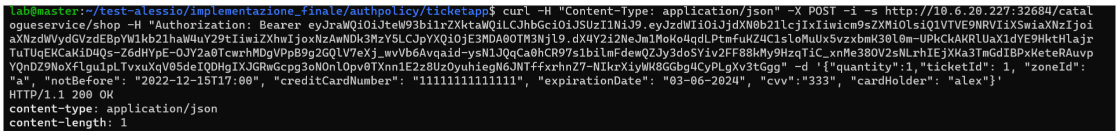
Figure 7.9: Test 4



```
lab@master:~$ kubectl exec -it psql-pod-6f7d649b5-sjp7z -n ticketapp -- psql -h traveler-postgres.ticketapp -p 5432 -U postgres postgres
psql: error: connection to server at "traveler-postgres.ticketapp" (10.107.47.186), port 5432 failed: server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
command terminated with exit code 2
```

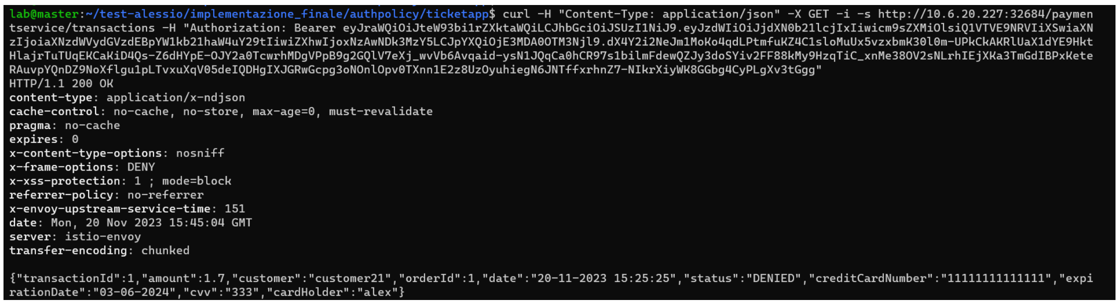
Figure 7.10: Test 5

Finally, in order to prove that communications between the various microservices and Kafka, as well as that between Kakfa and Zookeper are also authorised, I carried out a further test by contacting the /shop endpoint of the Catalogue service from outside the cluster via the Ingress Gateway (as this was the only way to contact the service due to the policies applied): Figure 7.11 shows the /shop request sent using a valid JWT and the positive response *200 OK*. In order to verify that indeed the communication between Catalogue and the services was correct, I then went to contact the /transactions endpoint of the Payment Service: this endpoint returns the list of transactions attempted by the Payment service and their outcome, and thus, given the presence of a transaction stored in the service's DB would necessarily imply that a purchase request was received by the Catalogue and processed. Figure 7.12 shows how indeed a transaction has been attempted, and thus that the service was contacted by Catalogue to handle the payment operation, which was then denied and stored. This not only shows that Payment is also able to access its postgres database to store and retrieve the requests data, but more importantly that the communication via Kafka works and consequently so does the Kafka-Zookeeper communication, otherwise the communication between the two microservices would never have been possible without Zookeeper as well.



```
lab@master:~/test-alessio/implementazione_finale/authpolicy/ticketapp$ curl -H "Content-Type: application/json" -X POST -i -s http://10.6.20.227:32684/catal
ogueservice/shop -H "Authorization: Bearer eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwicm9sZXMiOlsiQ1VTVE9NRVIiXSwiaXNzIjoi
aXNzdWVydGVzdEBpYW1kb21haW4uY29tIiwiZXhwIjoxNzAwNDk3MzY5LCJpYXQiOjE3MDA0OTM3Njl9.dX4Y2i2NeJm1MoKo4qdLPtmfuKZ4C1sloMuUx5vzxbmK30l0m-UPkCkAKRlUaX1dYE9HktHlajr
TuTUqEKCaKiD4Qs-Z6dHYpE-OJY2a0TcwrhMDgVPpB9g2GQlV7eXj_wvVb6Avqaid-ysN1JQqCa0hCR97s1bilmFdewQZJy3doSYiv2FF88kMy9HzqTiC_xnMe38OV2sNLrhIEjXKa3TmGdIBPxKeteRAuvp
YQnDZ9NoXflgu1pLTvxuXqV05deIQDHgIXJGRwGcpg3oNOnlOpv0TXnn1E2z8UzOyuhiegN6JNTffxrhnZ7-NIkrXiyWK8GGbg4CyPLgXv3tGgg" -d '{"quantity":1,"ticketId": 1, "zoneId":
"a", "notBefore": "2022-12-15T17:00", "creditCardNumber": "11111111111111", "expirationDate": "03-06-2024", "cvv":"333", "cardHolder": "alex"}'
HTTP/1.1 200 OK
content-type: application/json
content-length: 1
```

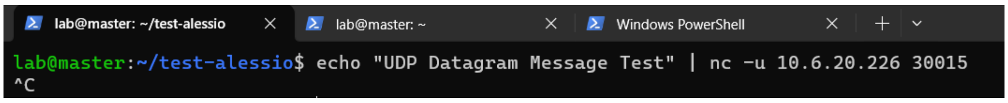
Figure 7.11: Test 6



```
lab@master:~/test-alessio/implementazione_finale/authpolicy/ticketapp$ curl -H "Content-Type: application/json" -X GET -i -s http://10.6.20.227:32684/paymen
tservice/transactions -H "Authorization: Bearer eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwicm9sZXMiOlsiQ1VTVE9NRVIiXSwiaXN
zIjoiaXNzdWVydGVzdEBpYW1kb21haW4uY29tIiwiZXhwIjoxNzAwNDk3MzY5LCJpYXQiOjE3MDA0OTM3Njl9.dX4Y2i2NeJm1MoKo4qdLPtmfuKZ4C1sloMuUx5vzxbmK30l0m-UPkCkAKRlUaX1dYE9Hkt
HlajrTuTUqEKCaKiD4Qs-Z6dHYpE-OJY2a0TcwrhMDgVPpB9g2GQlV7eXj_wvVb6Avqaid-ysN1JQqCa0hCR97s1bilmFdewQZJy3doSYiv2FF88kMy9HzqTiC_xnMe38OV2sNLrhIEjXKa3TmGdIBPxKete
RAuvpYQnDZ9NoXflgu1pLTvxuXqV05deIQDHgIXJGRwGcpg3oNOnlOpv0TXnn1E2z8UzOyuhiegN6JNTffxrhnZ7-NIkrXiyWK8GGbg4CyPLgXv3tGgg"
HTTP/1.1 200 OK
content-type: application/x-ndjson
cache-control: no-cache, no-store, max-age=0, must-revalidate
pragma: no-cache
expires: 0
x-content-type-options: nosniff
x-frame-options: DENY
x-xss-protection: 1 ; mode=block
referrer-policy: no-referrer
x-envoy-upstream-service-time: 151
date: Mon, 20 Nov 2023 15:45:04 GMT
server: istio-envoy
transfer-encoding: chunked

{"transactionId":1,"amount":1.7,"customer":"customer21","orderId":1,"date":"20-11-2023 15:25:25","status":"DENIED","creditCardNumber":"11111111111111","expi
rationDate":"03-06-2024","cvv":"333","cardHolder":"alex"}
```

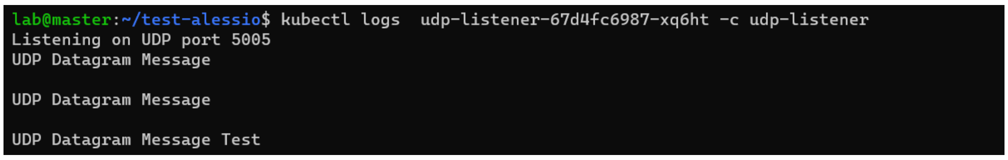
Figure 7.12: Test 7

In addition, another test was performed to prove that actually Envoy doesn't intercept non-TCP traffic, and thus it is not able to have visibility on it, and to enforce authentication and authorization policies. Thus I deployed an app that exposes a test UDP endpoint in the ticketapp namespace: so being an Istio namespace, that pod will be injected with a proxy envoy. In that namespace it is still applied an Authentication policy mTLS STRICT and an Authorization policy DENY ALL as done for the 3 namespaces in the previous tests: The UDP service is exposed via nodeport on port 30015, and I then go to send via the netcat tool a UDP packet to that service to see if the proxy intercepts it. Figure 7.13 shows the last UDP message sent to the service, while Figure 7.14 shows the various UDP messages received by the service in the various tests done, of which the last message is the last one sent in the test done in Figure 7.13. This proves that non-TCP traffic is not in fact intercepted and therefore authorized. In addition also ICMP protocol was tested with ping over pods IP, and also this kind of test failed, leaving the ICMP packets leave the pod: Figure 7.15 shows the ping performed by the Catalogue service towards the Traveler service pod IP freely flows without any problem.



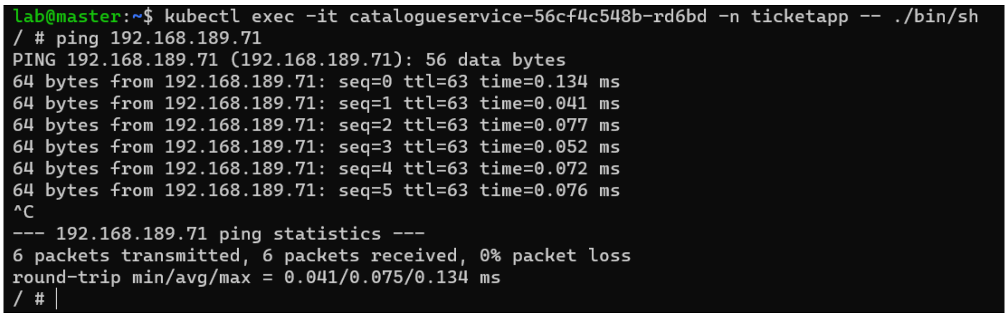
```
lab@master:~/test-alessio$ echo "UDP Datagram Message Test" | nc -u 10.6.20.226 30015
^C
```

Figure 7.13: Test UDP - source view



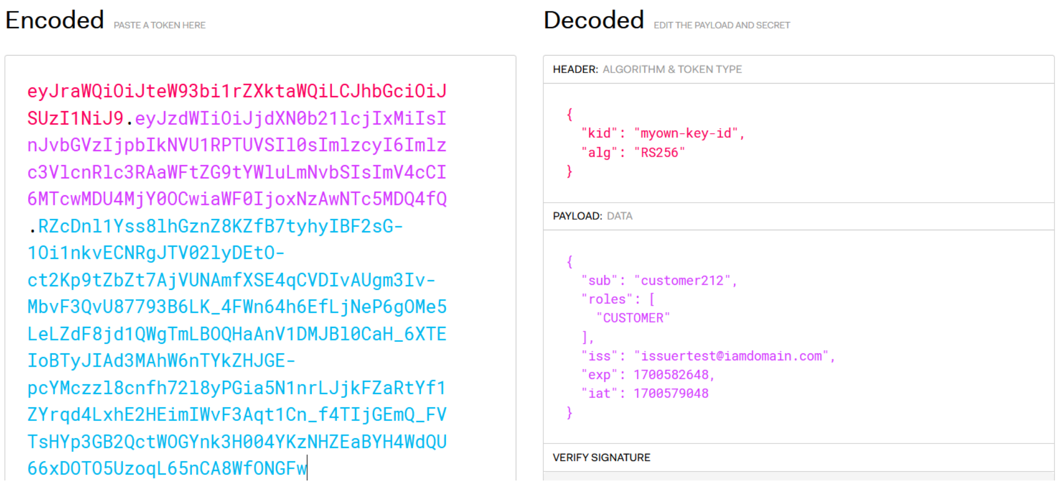
```
lab@master:~/test-alessio$ kubectl logs  udp-listener-67d4fc6987-xq6ht -c udp-listener
Listening on UDP port 5005
UDP Datagram Message

UDP Datagram Message

UDP Datagram Message Test
```

Figure 7.14: Test UDP - destination view



```
lab@master:~$ kubectl exec -it catalogueservice-56cf4c548b-rd6bd -n ticketapp -- ./bin/sh
/ # ping 192.168.189.71
PING 192.168.189.71 (192.168.189.71): 56 data bytes
64 bytes from 192.168.189.71: seq=0 ttl=63 time=0.134 ms
64 bytes from 192.168.189.71: seq=1 ttl=63 time=0.041 ms
64 bytes from 192.168.189.71: seq=2 ttl=63 time=0.077 ms
64 bytes from 192.168.189.71: seq=3 ttl=63 time=0.052 ms
64 bytes from 192.168.189.71: seq=4 ttl=63 time=0.072 ms
64 bytes from 192.168.189.71: seq=5 ttl=63 time=0.076 ms
^C
--- 192.168.189.71 ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 0.041/0.075/0.134 ms
/ # 
```

Figure 7.15: Test ICMP

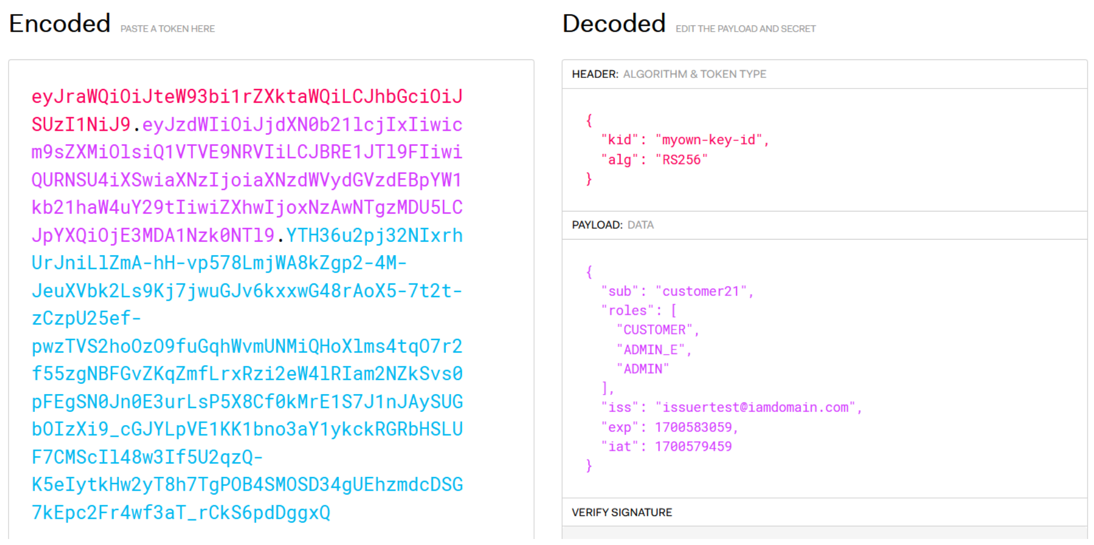


```
eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJ
SUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxMiIsI
nJvbGVzIjpbIkNVU1RPTUVSIl0sImlzcyI6Imlz
c3VlcnRlc3RAaWFtZG9tYWluLmNvbSIsImV4cCI
6MTcwMDU4MjY0OCwiaWF0IjoxNzAwNTc5MDQ4fQ
.RZcDnl1Yss8lhGznZ8KZfB7tyhyIBF2sG-
1Oi1nkvECNRgJTV02lyDEtO-
ct2Kp9tZbZt7AjVUNAmfXSE4qCVDIvAUgm3Iv-
MbvF3QvU87793B6LK_4FWn64h6EfLjNeP6gOMe5
LeLZdF8jd1QWgTmLBOQHaAnV1DMJBl0CaH_6XTE
IoBTyJIAd3MAhW6nTYkZHJGE-
pcYMczzl8cnfh72l8yPGia5N1nrLJjkFZaRtYf1
ZYrqd4LxhE2HEimIWvF3Aqt1Cn_f4TIjGEmQ_FV
TsHYp3GB2QctWOGYnk3H004YKzNHZEaBYH4WdQU
66xDOTO5UzoqL65nCA8WfONGFw
```

```
{
  "kid": "myown-key-id",
  "alg": "RS256"
}
```

```
{
  "sub": "customer212",
  "roles": [
    "CUSTOMER"
  ],
  "iss": "issuertest@iamdomain.com",
  "exp": 1700582648,
  "iat": 1700579048
}
```

Figure 7.16: Customer JWT



```
eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJ
SUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwic
m9sZXMiOlsiQ1VTVE9NRVIiLCJBRE1JTl9FIiwi
QURNSU4iXSwiaXNzIjoiaXNzdWVydGVzdEBpYW1
kb21haW4uY29tIiwiZXhwIjoxNzAwNTgzMDU5LC
JpYXQiOjE3MDA1Nzk0NTl9.YTH36u2pj32NIxrh
UrJniLlZmA-hH-vp578LmjWA8kZgp2-4M-
JeuXVbk2Ls9Kj7jwuGJv6kxxwG48rAoX5-7t2t-
zCzpU25ef-
pwzTVS2hoOzO9fuGqhWvmUNMiQHoXlms4tqO7r2
f55zgNBFGvZKqZmfLrxRzi2eW4lRIam2NZkSvs0
pFEgSN0Jn0E3urLsP5X8Cf0kMrE1S7J1nJAySUG
bOIzXi9_cGJYLpVE1KK1bno3aY1ykckRGRbHSLU
F7CMScIl48w3If5U2qzQ-
K5eIytkHw2yT8h7TgPOB4SMOSD34gUEhzmdcDSG
7kEpc2Fr4wf3aT_rCkS6pdDggxQ
```

```
{
  "kid": "myown-key-id",
  "alg": "RS256"
}
```

```
{
  "sub": "customer21",
  "roles": [
    "CUSTOMER",
    "ADMIN_E",
    "ADMIN"
  ],
  "iss": "issuertest@iamdomain.com",
  "exp": 1700583059,
  "iat": 1700579459
}
```

Figure 7.17: Admin JWT

7.1.2 End-User Authentication And Authorization

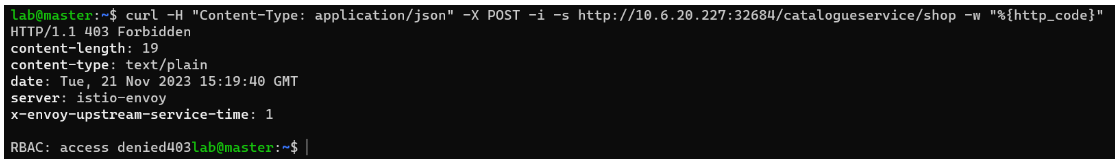
In order to check whether the End-User Authentication and Authorization policies work correctly, I performed several tests in which I sent several requests to my microservices from outside the cluster (due to the other policies applied in section 6.1.1), i.e. via the Ingress Gateway.

In particular, in order to also be able to verify the check carried out on the endpoints */admin/** and the respective user role possessed in the JWT, I logged in using two different users, one having an admin role, and the other not. Figure 7.16 shows the JWT of the user with the ADMIN role, while Figure 7.17 shows the one without (having only the CUSTOMER role). Both JWT are valid (neither expired nor with an invalid signature) and with a valid issuer embedded.

Figures 7.19 and 7.18 show the two tests performed on the Catalogue server. In particular, I went to send an HTTP POST /shop request to the service, both with and without JWT. This endpoint requires prior authentication of the user, in order to be able to subsequently manage their purchase data, tickets, and personal information. This is why the JWT is strictly necessary. Figure 7.18 shows how indeed without any JWT inserted in the authorisation header, the request is denied, receiving a *403 - Forbidden* reply and an *RBAC: access denied* message, a sign that the proxy authentication has indeed been bypassed (otherwise a *401 Unauthorized* message would have been returned), but PEP has blocked it as, since the JWT and therefore the claim field *sub* was absent, it was unable to retrieve the requestPrincipal.

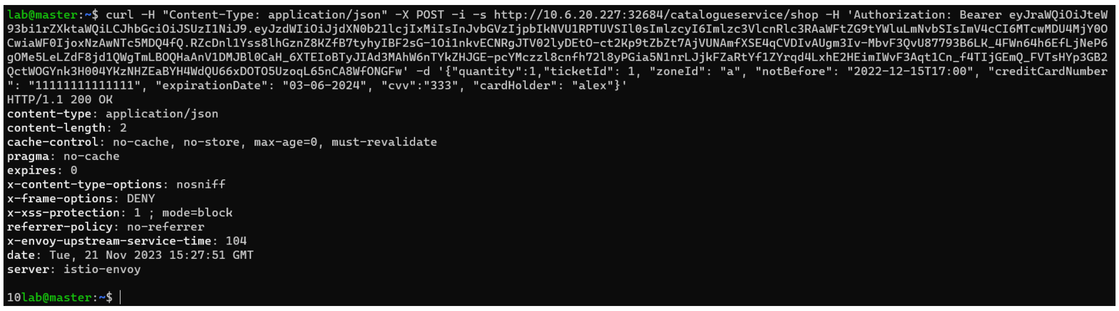
Figure 7.19 instead shows how using a valid JWT (even without the ADMIN role), the request succeeded, returning a *200 OK* and the order number taken, *10*. So given that the request was authorized, this means that first of all the mTLS peer authentication was successful and therefore the traffic was protected, subsequently the authentication of the JWT was also successful (and therefore the proxy correctly downloaded the public key from the provided endpoint and verified the signature, its integrity and temporal validity, and that its issuer was correct), and finally that the principal client (i.e. the client identity extracted from the certificate), was indeed the Ingress Gateway. Also note that in the case of the request without JWT, the DENY policy is evaluated before the other ALLOWs for a specific workload, so before the ALLOW defined in section 6.1.1

can match, the former will match and deny the traffic.



```
lab@master:~$ curl -H "Content-Type: application/json" -X POST -i -s http://10.6.20.227:32684/catalogueservice/shop -w "%{http_code}"
HTTP/1.1 403 Forbidden
content-length: 19
content-type: text/plain
date: Tue, 21 Nov 2023 15:19:40 GMT
server: istio-envoy
x-envoy-upstream-service-time: 1

RBAC: access denied403lab@master:~$ |
```

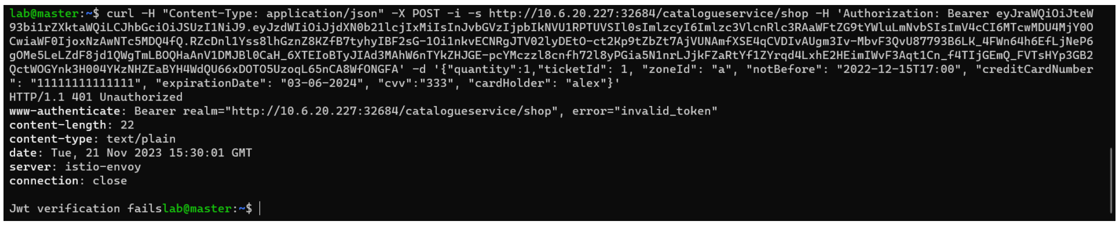
Figure 7.18: Test without JWT



```
lab@master:~$ curl -H "Content-Type: application/json" -X POST -i -s http://10.6.20.227:32684/catalogueservice/shop -H 'Authorization: Bearer eyJraWQiOiJteW
93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxMiIsInJvbGVzIjpbIkNVU1RPTUVSIl0sImlzcyI6Imlzc3VlcnRlc3RAaWFtZG9tYWluLmNvbSIsImV4cCI6MTcwMDU4MjY0O
CwiaWF0IjoxNzAwNTc5MDQ4fQ.RZcDnl1Yss8lhGznZ8KZfB7tyhyIBF2sG-1Oi1nkvECNRgJTV02lyDEtO-ct2Kp9tZbZt7AjVUNAmfXSE4qCVDIvAUgm3Iv-MbvF3QvU87793B6LK_4FWn64h6EfLjNeP6
gOMe5LeLZdF8jd1QWgTmLBOQHaAnV1DMJBl0CaH_6XTEIoBTyJIAd3MAhW6nTYkZHJGE-pcYMczzl8cnfh72l8yPGia5N1nrLJjkFZaRtYf1ZYrqd4LxhE2HEimIWvF3Aqt1Cn_f4TIjGEmQ_FVTsHYp3GB2
QctWOGYnk3H004YKzNHZEaBYH4WdQU66xDOTO5UzoqL65nCA8WfONGFw' -d '{"quantity":1,"ticketId": 1, "zoneId": "a", "notBefore": "2022-12-15T17:00", "creditCardNumber
": "1111111111111", "expirationDate": "03-06-2024", "cvv":"333", "cardHolder": "alex"}'
HTTP/1.1 200 OK
content-type: application/json
content-length: 2
cache-control: no-cache, no-store, max-age=0, must-revalidate
pragma: no-cache
expires: 0
x-content-type-options: nosniff
x-frame-options: DENY
x-xss-protection: 1 ; mode=block
referrer-policy: no-referrer
x-envoy-upstream-service-time: 104
date: Tue, 21 Nov 2023 15:27:51 GMT
server: istio-envoy

10lab@master:~$ |
```

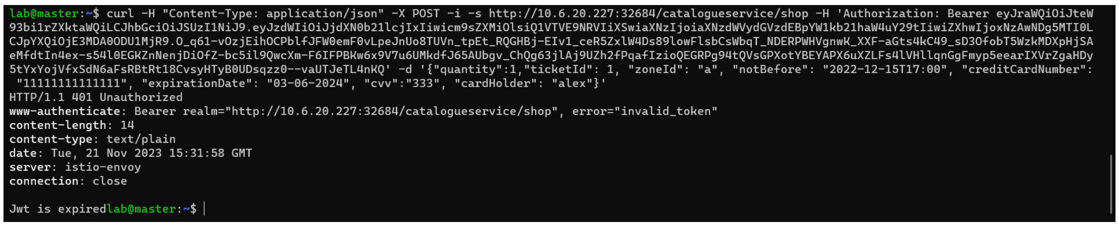
Figure 7.19: Test with valid JWT

In figures 7.21 and 7.20 instead, I repeated the same test at the shop endpoint, but first using a token modified from the original (thus compromised, 7.20), and then a second time using a token with a valid, but expired signature (7.21). Note that both return a *401 - Unauthorised* but different messages: *Jwt verification fails* the first and *Jwt is expired* the second one, confirming the reasons why the proxy denied the request.



```
lab@master:~$ curl -H "Content-Type: application/json" -X POST -i -s http://10.6.20.227:32684/catalogueservice/shop -H 'Authorization: Bearer eyJraWQiOiJteW
93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxMiIsInJvbGVzIjpbIkNVU1RPTUVSIl0sImlzcyI6Imlzc3VlcnRlc3RAaWFtZG9tYWluLmNvbSIsImV4cCI6MTcwMDU4MjY0O
CwiaWF0IjoxNzAwNTc5MDQ4fQ.RZcDnl1Yss8lhGznZ8KZfB7tyhyIBF2sG-1Oi1nkvECNRgJTV02lyDEtO-ct2Kp9tZbZt7AjVUNAmfXSE4qCVDIvAUgm3Iv-MbvF3QvU87793B6LK_4FWn64h6EfLjNeP6
gOMe5LeLZdF8jd1QWgTmLBOQHaAnV1DMJBl0CaH_6XTEIoBTyJIAd3MAhW6nTYkZHJGE-pcYMczzl8cnfh72l8yPGia5N1nrLJjkFZaRtYf1ZYrqd4LxhE2HEimIWvF3Aqt1Cn_f4TIjGEmQ_FVTsHYp3GB2
QctWOGYnk3H004YKzNHZEaBYH4WdQU66xDOTO5UzoqL65nCA8WfONGFA' -d '{"quantity":1,"ticketId": 1, "zoneId": "a", "notBefore": "2022-12-15T17:00", "creditCardNumber
": "11111111111111", "expirationDate": "03-06-2024", "cvv":"333", "cardHolder": "alex"}'
HTTP/1.1 401 Unauthorized
www-authenticate: Bearer realm="http://10.6.20.227:32684/catalogueservice/shop", error="invalid_token"
content-length: 22
content-type: text/plain
date: Tue, 21 Nov 2023 15:30:01 GMT
server: istio-envoy
connection: close

Jwt verification failslab@master:~$ |
```

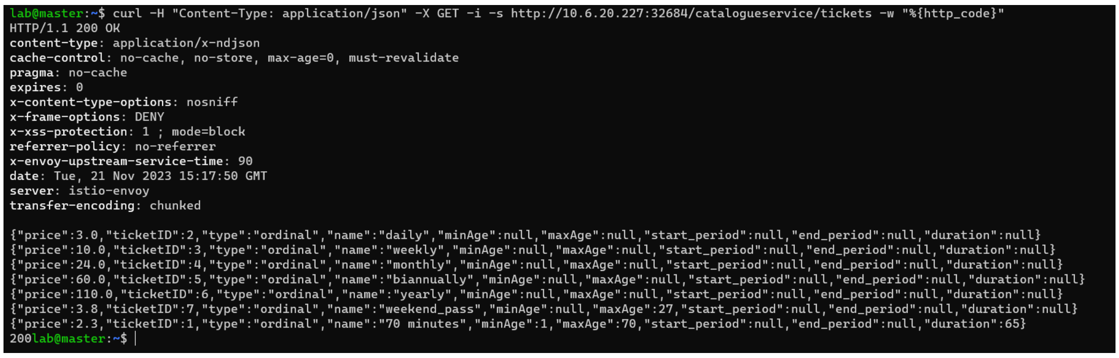
Figure 7.20: Test with expired JWT



```
lab@master:~$ curl -H "Content-Type: application/json" -X POST -i -s http://10.6.20.227:32684/catalogueservice/shop -H 'Authorization: Bearer eyJraWQiOiJteW
93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwicm9sZXMiOlsiQ1VTVE9NRVIiXSwiaXNzIjoiaXNzdWVydGVzdEBpYW1kb21haW4uY29tIiwiZXhwIjoxNzAwNDg5MTI0L
CJpYXQiOjE3MDA0ODU1MjR9.O_q61-vOzjEihOCPblfJFW0emF0vLpeJnUo8TUVn_tpEt_RQGHBj-EIv1_ceR5ZxlW4Ds89lowFlsbCsWbqT_NDERPWHVgnwK_XXF-aGts4kC49_sD3OfobT5WzkMDXpHjSA
eMfdtIn4ex-s54l0EGKZnNenjDiOfZ-bc5il9QwcXm-F6IFPBKw6x9V7u6UMkdfJ65AUbgv_ChQg63jlAj9UZh2fPqafIzioQEGRPg94tQVsGPXotYBEYAPX6uXZLFs4lVHllqnGgFmyp5eearIXVrZgaHDy
5tYxYojVfxSdN6aFsRBtRt18CvsyHTyB0UDsqzz0--vaUTJeTL4nKQ' -d '{"quantity":1,"ticketId": 1, "zoneId": "a", "notBefore": "2022-12-15T17:00", "creditCardNumber":
 "11111111111111", "expirationDate": "03-06-2024", "cvv":"333", "cardHolder": "alex"}'
HTTP/1.1 401 Unauthorized
www-authenticate: Bearer realm="http://10.6.20.227:32684/catalogueservice/shop", error="invalid_token"
content-length: 14
content-type: text/plain
date: Tue, 21 Nov 2023 15:31:58 GMT
server: istio-envoy
connection: close

Jwt is expiredlab@master:~$ |
```

Figure 7.21: Test with tampered JWT

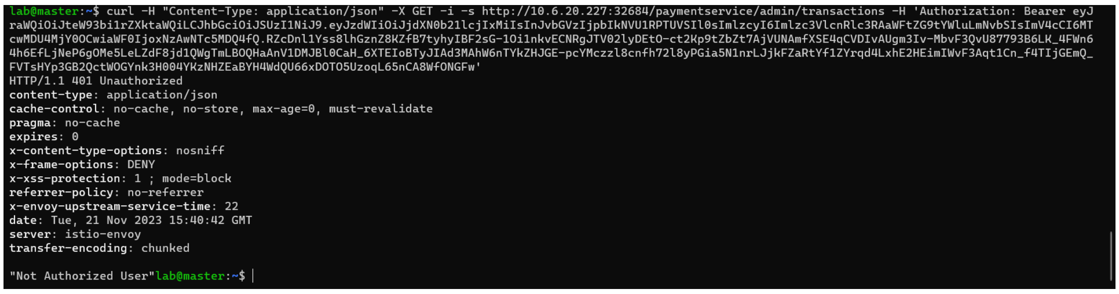
Figure 7.22 shows how instead, the endpoint /tickets is freely accessible without any JWT, as intended.



```
lab@master:~$ curl -H "Content-Type: application/json" -X GET -i -s http://10.6.20.227:32684/catalogueservice/tickets -w "%{http_code}"
HTTP/1.1 200 OK
content-type: application/x-ndjson
cache-control: no-cache, no-store, max-age=0, must-revalidate
pragma: no-cache
expires: 0
x-content-type-options: nosniff
x-frame-options: DENY
x-xss-protection: 1 ; mode=block
referrer-policy: no-referrer
x-envoy-upstream-service-time: 90
date: Tue, 21 Nov 2023 15:17:50 GMT
server: istio-envoy
transfer-encoding: chunked

{"price":3.0,"ticketID":2,"type":"ordinal","name":"daily","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":10.0,"ticketID":3,"type":"ordinal","name":"weekly","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":24.0,"ticketID":4,"type":"ordinal","name":"monthly","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":60.0,"ticketID":5,"type":"ordinal","name":"biannually","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":110.0,"ticketID":6,"type":"ordinal","name":"yearly","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":3.8,"ticketID":7,"type":"ordinal","name":"weekend_pass","minAge":null,"maxAge":27,"start_period":null,"end_period":null,"duration":null}
{"price":2.3,"ticketID":1,"type":"ordinal","name":"70 minutes","minAge":1,"maxAge":70,"start_period":null,"end_period":null,"duration":65}
200lab@master:~$ |
```

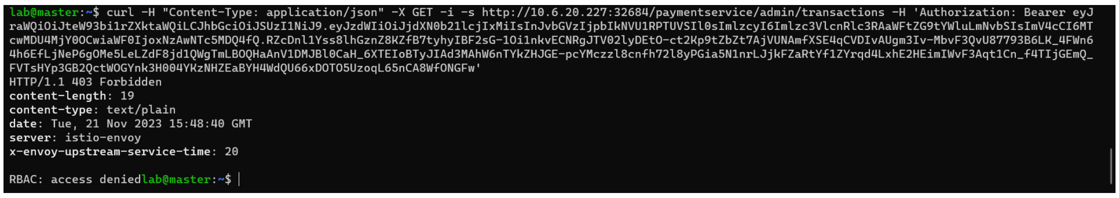
Figure 7.22: Test /ticktes endpoint

Finally, I tested the Role Based Access Control on the ADMIN role when accessing privileged endpoints: Figure 7.23 shows how before applying this authorization, by contacting the Payment service */admin/transactions* endpoint and providing a valid JWT but with only the CUSTOMER role I was able to reach the application layer of my service, but then the JWT controls implemented at the software level by me still denied the request (returning the message *Not Authorised User*, and instead after applying the policy on the /admin endpoints, the same request (Figure 7.24) is still denied, but by the proxy itself at the "infrastructure" level, thus preventing it from reaching the application layer of my container and returning the message *RBAC: access denied.*



```
lab@master:~$ curl -H "Content-Type: application/json" -X GET -i -s http://10.6.20.227:32684/paymentservice/admin/transactions -H 'Authorization: Bearer eyJ
raWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxMiIsInJvbGVzIjpbIkNVU1RPTUVSIl0sImlzcyI6Imlzc3VlcnRlc3RAaWFtZG9tYWluLmNvbSIsImV4cCI6MT
cwMDU4MjY0OCwiaWF0IjoxNzAwNTc5MDQ4fQ.RZcDnl1Yss8lhGznZ8KZfB7tyhyIBF2sG-1Oi1nkvECNRgJTV02lyDEtO-ct2Kp9tZbZt7AjVUNAmfXSE4qCVDIvAUgm3Iv-MbvF3QvU87793B6LK_4FWn6
4h6EfLjNeP6gOMe5LeLZdF8jd1QWgTmLBOQHaAnV1DMJBl0CaH_6XTEIoBTyJIAd3MAhW6nTYkZHJGE-pcYMczzl8cnfh72l8yPGia5N1nrLJjkFZaRtYf1ZYrqd4LxhE2HEimIWvF3Aqt1Cn_f4TIjGEmQ_
FVTsHYp3GB2QctWOGYnk3H004YKzNHZEaBYH4WdQU66xDOTO5UzoqL65nCA8WfONGFw'
HTTP/1.1 401 Unauthorized
content-type: application/json
cache-control: no-cache, no-store, max-age=0, must-revalidate
pragma: no-cache
expires: 0
x-content-type-options: nosniff
x-frame-options: DENY
x-xss-protection: 1 ; mode=block
referrer-policy: no-referrer
x-envoy-upstream-service-time: 22
date: Tue, 21 Nov 2023 15:40:42 GMT
server: istio-envoy
transfer-encoding: chunked

"Not Authorized User"lab@master:~$ |
```

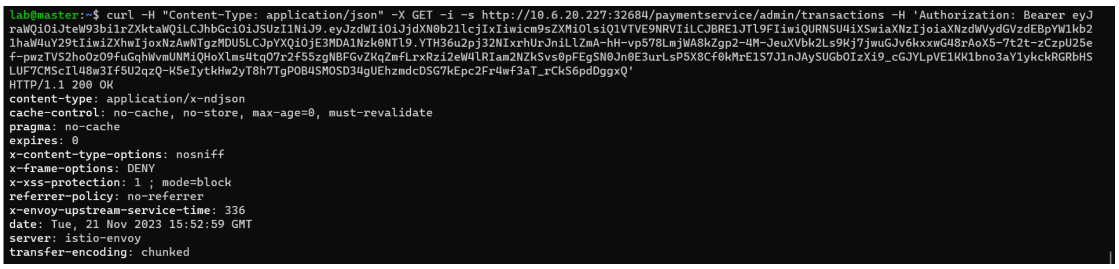
Figure 7.23: Test Admin Endpoint 1



```
lab@master:~$ curl -H "Content-Type: application/json" -X GET -i -s http://10.6.20.227:32684/paymentservice/admin/transactions -H 'Authorization: Bearer eyJ
raWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxMiIsInJvbGVzIjpbIkNVU1RPTUVSIl0sImlzcyI6Imlzc3VlcnRlc3RAaWFtZG9tYWluLmNvbSIsImV4cCI6MT
cwMDU4MjY0OCwiaWF0IjoxNzAwNTc5MDQ4fQ.RZcDnl1Yss8lhGznZ8KZfB7tyhyIBF2sG-1Oi1nkvECNRgJTV02lyDEtO-ct2Kp9tZbZt7AjVUNAmfXSE4qCVDIvAUgm3Iv-MbvF3QvU87793B6LK_4FWn6
4h6EfLjNeP6gOMe5LeLZdF8jd1QWgTmLBOQHaAnV1DMJBl0CaH_6XTEIoBTyJIAd3MAhW6nTYkZHJGE-pcYMczzl8cnfh72l8yPGia5N1nrLJjkFZaRtYf1ZYrqd4LxhE2HEimIWvF3Aqt1Cn_f4TIjGEmQ_
FVTsHYp3GB2QctWOGYnk3H004YKzNHZEaBYH4WdQU66xDOTO5UzoqL65nCA8WfONGFw'
HTTP/1.1 403 Forbidden
content-length: 19
content-type: text/plain
date: Tue, 21 Nov 2023 15:48:40 GMT
server: istio-envoy
x-envoy-upstream-service-time: 20

RBAC: access deniedlab@master:~$ |
```

Figure 7.24: Test Admin Endpoint 2

Instead, repeating the test after applying the policy but with a JWT containing the ADMIN role (figure 7.25), my request is successful and I am able to get back a list of transactions from the service.

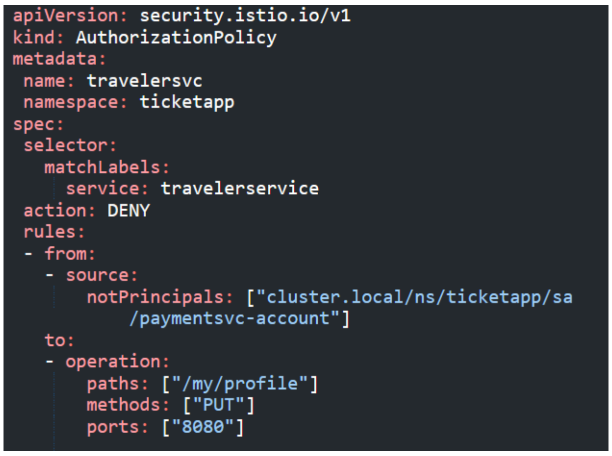


```
lab@master:~$ curl -H "Content-Type: application/json" -X GET -i -s http://10.6.20.227:32684/paymentservice/admin/transactions -H 'Authorization: Bearer eyJ
raWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwicm9sZXMiOlsiQ1VTVE9NRVIiLCJBRE1JTl9FIiwiQURNSU4iXSwiaXNzIjoiaXNzdWVydGVzdEBpYW1kb2
1haW4uY29tIiwiZXhwIjoxNzAwNTgzMDU5LCJpYXQiOjE3MDA1Nzk0NTl9.YTH36u2pj32NIxrhUrJniLlZmA-hH-vp578LmjWA8kZgp2-4M-JeuXVbk2Ls9Kj7jwuGJv6kxxwG48rAoX5-7t2t-zCzpU25e
f-pwzTVS2hoOzO9fuGqhWvmUNMiQHoXlms4tqO7r2f55zgNBFGvZKqZmfLrxRzi2eW4lRIam2NZkSvs0pFEgSN0Jn0E3urLsP5X8Cf0kMrE1S7J1nJAySUGbOIzXi9_cGJYLpVE1KK1bno3aY1ykckRGRbHS
LUF7CMScIl48w3If5U2qzQ-K5eIytkHw2yT8h7TgPOB4SMOSD34gUEhzmdcDSG7kEpc2Fr4wf3aT_rCkS6pdDggxQ'
HTTP/1.1 200 OK
content-type: application/x-ndjson
cache-control: no-cache, no-store, max-age=0, must-revalidate
pragma: no-cache
expires: 0
x-content-type-options: nosniff
x-frame-options: DENY
x-xss-protection: 1 ; mode=block
referrer-policy: no-referrer
x-envoy-upstream-service-time: 336
date: Tue, 21 Nov 2023 15:52:59 GMT
server: istio-envoy
transfer-encoding: chunked
```

Figure 7.25: Test Admin Endpoint 3

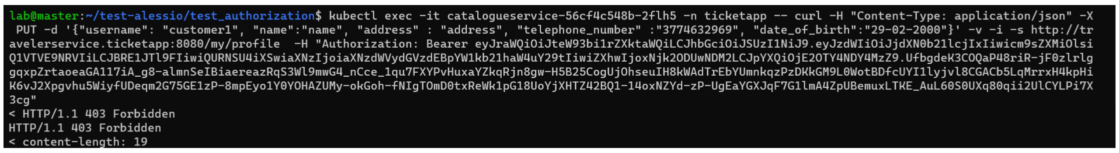
Although in the case of the microservice application I developed for testing, the various JWT checks (signature, expiration, RBAC) were already implemented at the software level for each microservice, having a security check at the infrastructure level instead of the software level facilitates its implementation, management and makes life easier for programmers by separating the security aspects from the development ones. The usage of JWT and claims within it to define policies, allow you leveraging on some specific attributes of the user (or the identity of the user itself) to authorize and microsegment the access to the resource. This, in combination with the workload-to-workload authorization thanks to the X.509 certificate, allows you to achieve a good level of zero trust in your network, especially for what concerning HTTP services.

Two further tests were carried out later to test the policy refinement offered by Istio. After removing the authorization policies of section 6.1.1 used to segment the expected traffic in the ticketing application, I assumed that the two endpoints of the same Traveler service *GET /my/profile* and *PUT /my/profile* had different security requirements. In particular, the first endpoint should be accessible from any workload with a valid identity (i.e. in the mesh), and thus without any authorisation of any kind, but only protected and authenticated traffic via mTLS; and instead the second, with the PUT method, should also have restricted access only to a specific workload, in this case for instance the Payment service, and thus a specific authorisation on a certain type of very precise requests (i.e. specific method and path). The Authorization Policy provided was the one in picture 7.26. Instead, the pictures 7.27 and 7.28 show respectively the Catalogue and the Payment service trying to contact the specific endpoint *PUT /my/profile*, and how in the first case the request was denied because the identity of the source workload is the Catalogue service account, while in the second case it was accepted without problems because it is the Payment service account. Figure 7.29, on the other hand, demonstrates how the Catalogue is free to contact the same workload path, but with different methods (e.g. GET).



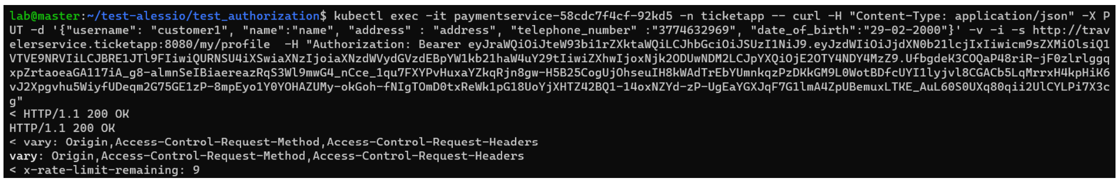
```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: travelersvc
  namespace: ticketapp
spec:
  selector:
    matchLabels:
      service: travelerservice
  action: DENY
  rules:
  - from:
    - source:
        notPrincipals: ["cluster.local/ns/ticketapp/sa
            /paymentsvc-account"]
    to:
    - operation:
        paths: ["/my/profile"]
        methods: ["PUT"]
        ports: ["8080"]
```

Figure 7.26: Specific Request Path And Method Authorization Policy



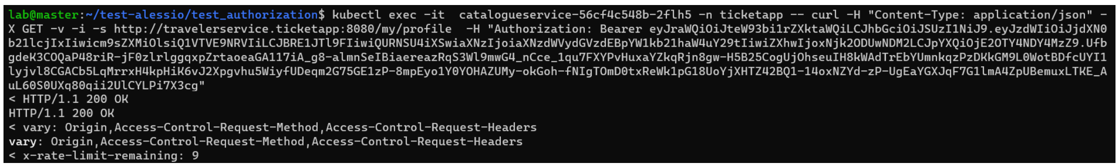
```
lab@master:~/test-alessio/test_authorization$ kubectl exec -it catalogueservice-56cf4c548b-2flh5 -n ticketapp -- curl -H "Content-Type: application/json" -X
 PUT -d '{"username": "customer1", "name":"name", "address" : "address", "telephone_number" :"3774632969", "date_of_birth":"29-02-2000"}' -v -i -s http://tr
avelerservice.ticketapp:8080/my/profile  -H "Authorization: Bearer eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwicm9sZXMiOlsi
Q1VTVE9NRVIiLCJBRE1JTl9FIiwiQURNSU4iXSwiaXNzIjoiaXNzdWVydGVzdEBpYW1kb21haW4uY29tIiwiZXhwIjoxNjk2ODUwNDM2LCJpYXQiOjE2OTY4NDY4MzZ9.UfbgdeK3COQaP48riR-jF0zlrlg
gqxpZrtaoeaGA117iA_g8-almnSeIBiaereazRqS3Wl9mwG4_nCce_1qu7FXYPvHuxaYZkqRjn8gw-H5B25CogUjOhseuIH8kWAdTrEbYUmnkqzPzDKkGM9L0WotBDfcUYI1lyjvl8CGACb5LqMrrxH4kpHi
K6vJ2Xpgvhu5WiyfUDeqm2G75GE1zP-8mpEyo1Y0YOHAZUMy-okGoh-fNIgTOmD0txReWk1pG18UoYjXHTZ42BQ1-14oxNZYd-zP-UgEaYGXJqF7G1lmA4ZpUBemuxLTKE_AuL60S0UXq80qii2UlCYLPi7X
3cg"
< HTTP/1.1 403 Forbidden
HTTP/1.1 403 Forbidden
< content-length: 19
```

Figure 7.27: Test 1 - Specific Request Path And Method Authorization Policy



```
lab@master:~/test-alessio/test_authorization$ kubectl exec -it paymentservice-58cdc7f4cf-92kd5 -n ticketapp -- curl -H "Content-Type: application/json" -X P
UT -d '{"username": "customer1", "name":"name", "address" : "address", "telephone_number" :"3774632969", "date_of_birth":"29-02-2000"}' -v -i -s http://trav
elerservice.ticketapp:8080/my/profile  -H "Authorization: Bearer eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxIiwicm9sZXMiOlsiQ1
VTVE9NRVIiLCJBRE1JTl9FIiwiQURNSU4iXSwiaXNzIjoiaXNzdWVydGVzdEBpYW1kb21haW4uY29tIiwiZXhwIjoxNjk2ODUwNDM2LCJpYXQiOjE2OTY4NDY4MzZ9.UfbgdeK3COQaP48riR-jF0zlrlggq
xpZrtaoeaGA117iA_g8-almnSeIBiaereazRqS3Wl9mwG4_nCce_1qu7FXYPvHuxaYZkqRjn8gw-H5B25CogUjOhseuIH8kWAdTrEbYUmnkqzPzDKkGM9L0WotBDfcUYI1lyjvl8CGACb5LqMrrxH4kpHiK6
vJ2Xpgvhu5WiyfUDeqm2G75GE1zP-8mpEyo1Y0YOHAZUMy-okGoh-fNIgTOmD0txReWk1pG18UoYjXHTZ42BQ1-14oxNZYd-zP-UgEaYGXJqF7G1lmA4ZpUBemuxLTKE_AuL60S0UXq80qii2UlCYLPi7X3c
g"
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
< x-rate-limit-remaining: 9
```

Figure 7.28: Test 2 - Specific Request Path And Method Authorization Policy



```
lab@master:~/test-alessio/test_authorization$ kubectl exec -it  catalogueservice-56cf4c548b-2flh5 -n ticketapp -- curl -H "Content-Type: application/json" -
X GET -v -i -s http://travelerservice.ticketapp:8080/my/profile  -H "Authorization: Bearer eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0
b21lcjIxIiwicm9sZXMiOlsiQ1VTVE9NRVIiLCJBRE1JTl9FIiwiQURNSU4iXSwiaXNzIjoiaXNzdWVydGVzdEBpYW1kb21haW4uY29tIiwiZXhwIjoxNjk2ODUwNDM2LCJpYXQiOjE2OTY4NDY4MzZ9.Ufb
gdeK3COQaP48riR-jF0zlrlggqxpZrtaoeaGA117iA_g8-almnSeIBiaereazRqS3Wl9mwG4_nCce_1qu7FXYPvHuxaYZkqRjn8gw-H5B25CogUjOhseuIH8kWAdTrEbYUmnkqzPzDKkGM9L0WotBDfcUYI1
lyjvl8CGACb5LqMrrxH4kpHiK6vJ2Xpgvhu5WiyfUDeqm2G75GE1zP-8mpEyo1Y0YOHAZUMy-okGoh-fNIgTOmD0txReWk1pG18UoYjXHTZ42BQ1-14oxNZYd-zP-UgEaYGXJqF7G1lmA4ZpUBemuxLTKE_A
uL60S0UXq80qii2UlCYLPi7X3cg"
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
< x-rate-limit-remaining: 9
```

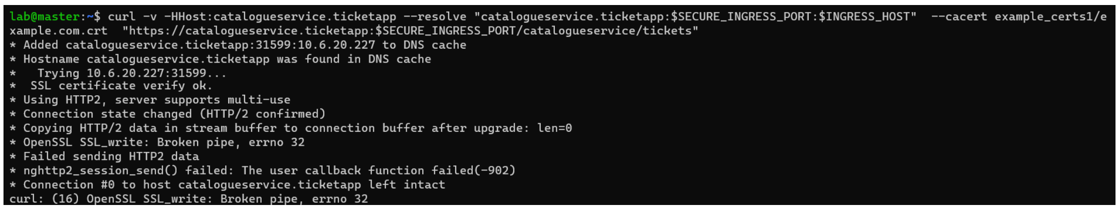
Figure 7.29: Test 3 - Specific Request Path And Method Authorization Policy

However, the fact that Istio allows authorization policies to be expressed and enforced using so many details within the rule specification means that it is easy to express increasingly fine-grained and workload-specific authorisation policies. Unfortunately, this level of detail in defining policies in Istio is only possible for HTTP traffic, since Service Mesh is primarily designed to optimise functionality and interactions between services of this type: Istio authorization supports any plain TCP protocols as well, but certain fields like and conditions are only applicable to HTTP workloads. *Hosts*, *methods* and *paths* fields, as well as the *requestPrincipal* one, can be used only with HTTP traffic. Thus the end-user authtentication and authorization can be performed only with that kind of traffic.

In any case, this potential of Istio can be exploited to fine-tune the type of authorization of each HTTP request, thus restricting access to certain endpoints of the various microservices to only certain workloads (using the identity in X.509 certificates) or to certain users (using the identity in either certificates or JWTs, or perhaps a combo of the two to provide greater security).

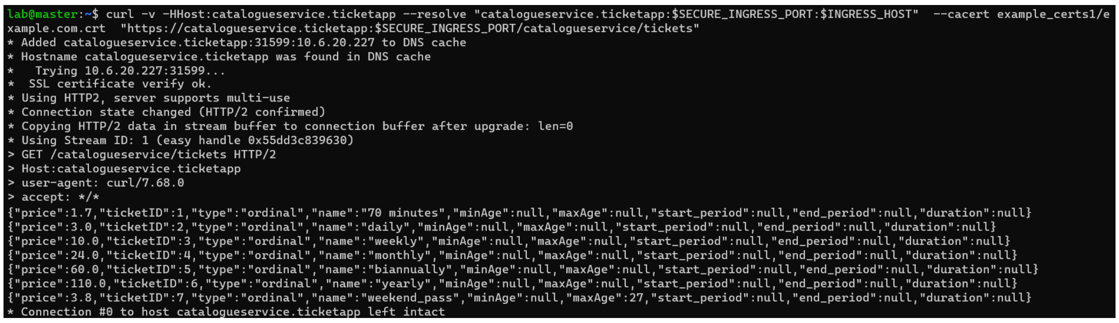
7.1.3 Ingress And Egress traffic

In order to test the Ingress Gateway functionalities, I leveraged *openssl* to first generate a custom CA private key and a root certificate, and subsequently I created a pair private key-certificate for each service inside the Mesh, that will be signed using the CA private key. In this way connections towards each service inside the Mesh will require a different and specific Service certificate for server authentication. Figures 7.30 and 7.31 show that in fact, when an X.509 certificate is not used, the TLS connection to the Catalogue service from outside the cluster is refused (fig. 7.30), ensuring that only after mutual authentication is possible: in fact, when using a valid certificate, the same endpoint is reachable and returns the response (fig. 7.31).



```
lab@master:~$ curl -v -HHost:catalogueservice.ticketapp --resolve "catalogueservice.ticketapp:$SECURE_INGRESS_PORT:$INGRESS_HOST"  --cacert example_certs1/e
xample.com.crt  "https://catalogueservice.ticketapp:$SECURE_INGRESS_PORT/catalogueservice/tickets"
* Added catalogueservice.ticketapp:31599:10.6.20.227 to DNS cache
* Hostname catalogueservice.ticketapp was found in DNS cache
*   Trying 10.6.20.227:31599...
*  SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
* OpenSSL SSL_write: Broken pipe, errno 32
* Failed sending HTTP2 data
* nghttp2_session_send() failed: The user callback function failed(-902)
* Connection #0 to host catalogueservice.ticketapp left intact
curl: (16) OpenSSL SSL_write: Broken pipe, errno 32
```

Figure 7.30: Ingress Test 1



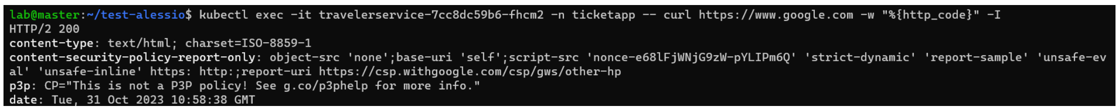
```
lab@master:~$ curl -v -HHost:catalogueservice.ticketapp --resolve "catalogueservice.ticketapp:$SECURE_INGRESS_PORT:$INGRESS_HOST"  --cacert example_certs1/e
xample.com.crt  "https://catalogueservice.ticketapp:$SECURE_INGRESS_PORT/catalogueservice/tickets"
* Added catalogueservice.ticketapp:31599:10.6.20.227 to DNS cache
* Hostname catalogueservice.ticketapp was found in DNS cache
*   Trying 10.6.20.227:31599...
*  SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
* Using Stream ID: 1 (easy handle 0x55dd3c839630)
> GET /catalogueservice/tickets HTTP/2
> Host:catalogueservice.ticketapp
> user-agent: curl/7.68.0
> accept: */*
{"price":1.7,"ticketID":1,"type":"ordinal","name":"70 minutes","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":3.0,"ticketID":2,"type":"ordinal","name":"daily","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":10.0,"ticketID":3,"type":"ordinal","name":"weekly","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":24.0,"ticketID":4,"type":"ordinal","name":"monthly","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":60.0,"ticketID":5,"type":"ordinal","name":"biannually","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":110.0,"ticketID":6,"type":"ordinal","name":"yearly","minAge":null,"maxAge":null,"start_period":null,"end_period":null,"duration":null}
{"price":3.8,"ticketID":7,"type":"ordinal","name":"weekend_pass","minAge":null,"maxAge":27,"start_period":null,"end_period":null,"duration":null}
* Connection #0 to host catalogueservice.ticketapp left intact
```

Figure 7.31: Ingress Test 2

Other tests concerning authorisation policies on authenticated and privileged endpoints were carried out and were basically the same as those carried out in Chapter 7.1.2. The results showed how indeed, the authentication of the peer, of the specific HTTP request, and the authorisation of the individual user and of the claims contained in the JWT attached to the request, can already be carried out by the ingress gateway, thus making it possible to anticipate any security checks, and thus reject unauthorised and/or authorised requests already at the edge of the cluster.

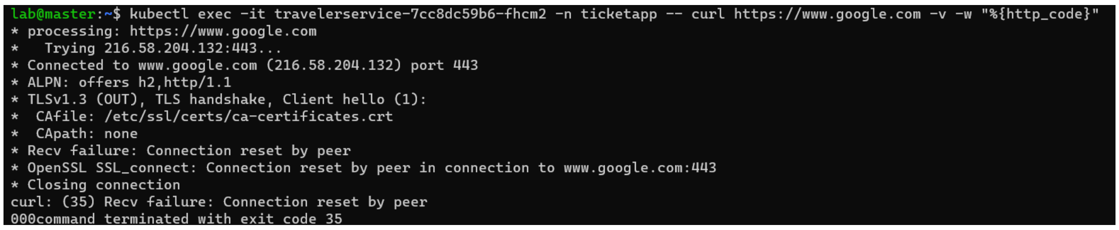
However, in order to implement a zero-trust approach, it is best to implement these authentication and authorisation policies (both peer and end-user) at the gateway level, but above all at the individual service level, because in order to guarantee an adequate security posture for the various services in the cluster, it is also important to protect against any unauthorised accesses made from within the corporate network and the cluster itself.

With regard to outgoing traffic, two tests were first performed (in the images 7.32 and 7.33), before and after configuring Istio in REGISTRY ONLY mode. The tests consisted of making the Traveler service contact "www.google.com" service. Note how indeed just before changing the configuration, traffic to random external services like Google (7.32) was let through, and afterwards, how the same traffic and request is blocked by the pod's proxy automatically (7.33), since the service 'www.google.com' is not present in Istio's service registry. Internal services within the mesh, on the other hand, being managed by Istio, have a Service Entry in the registry automatically, and thus let outgoing traffic flow to them from sources internal to the mesh itself (which is why server-side permissions are used to restrict intra-cluster communications afterwards). After deploying the VirtualService, ServiceEntry, Gateway and DestinationRule resources instead, now a path towards the Google service has been configured, and the proxies now forward the traffic for that DNS name to the egress gateway, and the egress gateway will forward it to the proper external service. Figure 7.34 shows how indeed, now the Traveler service now is able again to contact the Google service, but is still prevented from contacting all the others not explictly configured external services (i.e., connections towards "www.facebook.com" service, in Figure 7.35, has been blocked altough the Google one was permitted).



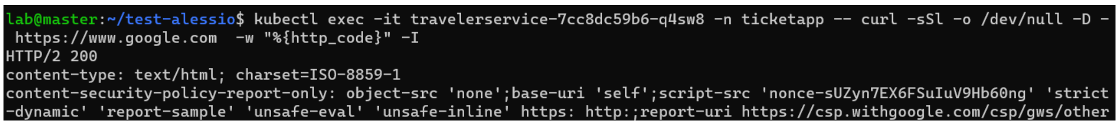
```
lab@master:~/test-alessio$ kubectl exec -it travelerservice-7cc8dc59b6-fhcm2 -n ticketapp -- curl https://www.google.com -w "%{http_code}" -I
HTTP/2 200
content-type: text/html; charset=ISO-8859-1
content-security-policy-report-only: object-src 'none';base-uri 'self';script-src 'nonce-e68lFjWNjG9zW-pYLIPm6Q' 'strict-dynamic' 'report-sample' 'unsafe-ev
al' 'unsafe-inline' https: http:;report-uri https://csp.withgoogle.com/csp/gws/other-hp
p3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."
date: Tue, 31 Oct 2023 10:58:38 GMT
```

Figure 7.32: Egress Test 1



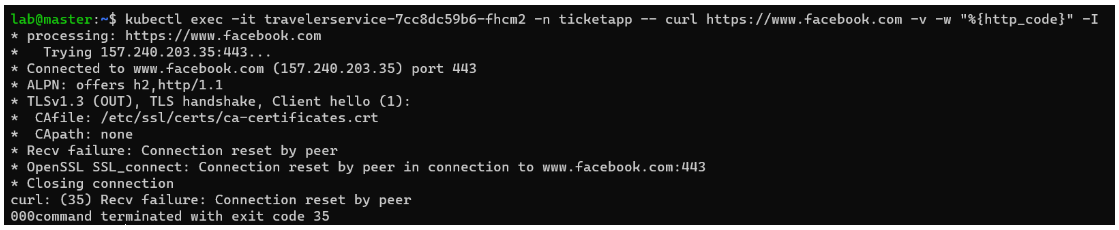
```
lab@master:~$ kubectl exec -it travelerservice-7cc8dc59b6-fhcm2 -n ticketapp -- curl https://www.google.com -v -w "%{http_code}"
* processing: https://www.google.com
*   Trying 216.58.204.132:443...
* Connected to www.google.com (216.58.204.132) port 443
* ALPN: offers h2,http/1.1
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
*  CAfile: /etc/ssl/certs/ca-certificates.crt
*  CApath: none
* Recv failure: Connection reset by peer
* OpenSSL SSL_connect: Connection reset by peer in connection to www.google.com:443
* Closing connection
curl: (35) Recv failure: Connection reset by peer
000command terminated with exit code 35
```

Figure 7.33: Egress Test 2



```
lab@master:~/test-alessio$ kubectl exec -it travelerservice-7cc8dc59b6-q4sw8 -n ticketapp -- curl -sSl -o /dev/null -D -
 https://www.google.com  -w "%{http_code}" -I
HTTP/2 200
content-type: text/html; charset=ISO-8859-1
content-security-policy-report-only: object-src 'none';base-uri 'self';script-src 'nonce-sUZyn7EX6FSuIuV9Hb60ng' 'strict
-dynamic' 'report-sample' 'unsafe-eval' 'unsafe-inline' https: http:;report-uri https://csp.withgoogle.com/csp/gws/other
```

Figure 7.34: Egress Test 3



```
lab@master:~$ kubectl exec -it travelerservice-7cc8dc59b6-fhcm2 -n ticketapp -- curl https://www.facebook.com -v -w "%{http_code}" -I
* processing: https://www.facebook.com
*   Trying 157.240.203.35:443...
* Connected to www.facebook.com (157.240.203.35) port 443
* ALPN: offers h2,http/1.1
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
*  CAfile: /etc/ssl/certs/ca-certificates.crt
*  CApath: none
* Recv failure: Connection reset by peer
* OpenSSL SSL_connect: Connection reset by peer in connection to www.facebook.com:443
* Closing connection
curl: (35) Recv failure: Connection reset by peer
000command terminated with exit code 35
```

Figure 7.35: Egress Test 4

So we we created an exit gateway for google.com, port 443: but it is possible to route multiple hosts through the egress gateway, by including a list of hosts, or using the "*" value for the hosts field in the gateway, to match them all.

However, while Ingress Gateway is useful for performing authentication at the edge of the cluster and preventing external unauthorized access to it, the Egress gateway is a symmetrical concept with respect to the former, and it defines exit points from the mesh. Also in this case it is useful to inspect the outgoing traffic leaving the Mesh, but also to force traffic leaving only to pass trough a specific node and be properly redirected.

Considering for example an organization that has strict security requirements, and thus that all traffic leaving the service mesh must flow through a set of dedicated nodes for policy enforcement and monitoring on the egress traffic (e.g a WAF), and these nodes run on dedicated machines, separated from the rest of the nodes running applications in the cluster: in such cases the egress gateway can be used in pair with Network Policies to ensure that the outgoing traffic of each wokload within the mesh can only pass through these security nodes, otherwise it will be blocked. Unfortunately, however, this is possible only for HTTP and HTTPS traffic, and thus this is a great limitation.

7.1.4 Network Traffic And Requests Visibility

However, Istio generates detailed telemetry for all service communications within a mesh, in order to provide overall service mesh workloads observability. To monitor services behavior, each proxy generates a rich set of metrics and distributed traces about all traffic passing through the proxy (both inbound and outbound). These metrics provide information on behaviors such as the overall volume of traffic, the error rates within the traffic, and the response times for the intercepted requests. Each proxy is also able to automatically generate access logs for each HTTP request, providing a way to monitor and understand behavior from the perspective of an individual workload instance.

Thus the presence of a proxy injected within each pod, is not only useful to enforce authentication and authorization policies, but also to gain information about the behaviour of each single workload, and their interaction with both internal and external services. This permits then to Envoy to easily send those data and logs to the Prometheus instance, the Istio addon that we installed together with Kiali and Grafana. Prometheus is a centralized monitoring system, which automatically collects and stores these data, while Kiali and Grafana will retrieve these stored data and represent them trough some specific Dahsboards: the former with a default Dashboard built on-purpose to show the most interesting data in Istio, the latter instead allows you to create custom dashboards as you prefer. This permits you to understand how traffic and requests flow among services, and to detect eventual anomalous behaviours by also setting on-purpose alerting systems integrated with these tools.

Figure 7.36 shows how Kiali Dashboard offers a powerful visualization of the mesh traffic, providing a Graph View of the requests and connection in a specified time interval. The various circle-shaped nodes are the workloads within the Mesh grouped by their own namespace, and the triangle-shaped nodes are the ClusterIP service to which the former belong, indicating whether they have been contacted through their ClusterIP address (and consequent resolution), or directly using the pod IP addresses (in case the triangle-shaped one is missing). Multiple Graph Types are provided, which allows you to visualize traffic as a high-level service topology, a low-level workload topology, or as an application-level topology. In the case of picture 7.36, this is a low-level workload one, showing specific workloads interactions. In particular, it is clear how communications are all mTLS protected from the "lock" symbol on each edge: edge colored in blue represents non-HTTP traffic (i.e., plain TCP traffic) that has been upgraded to a TLS one, while instead the edge representing HTTP requests (i.e., HTTP workload) appear with different colors depending on the response code returned to the various requests.

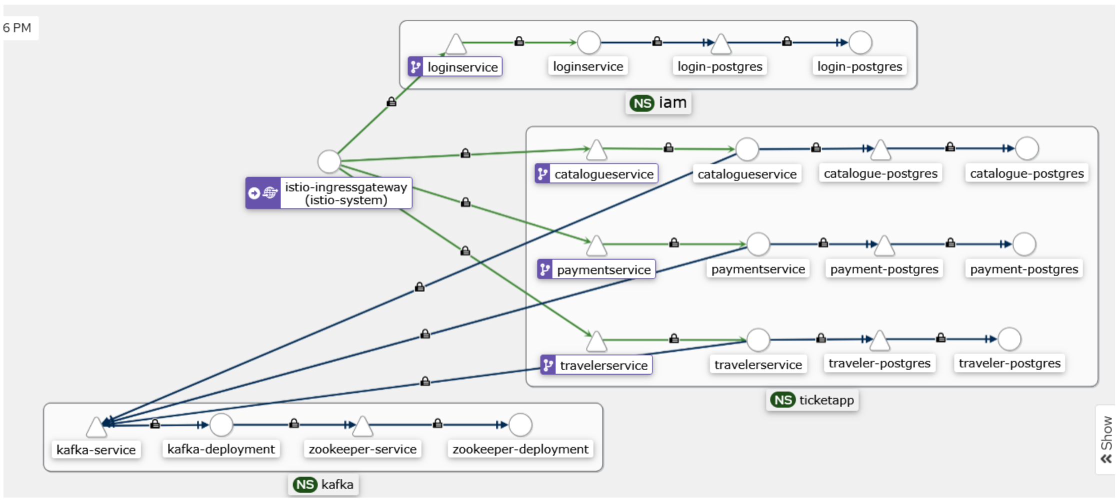


Figure 7.36: Kiali Dashboard Visbility 1

Figure 7.37 shows how Kiali also represents the identities of the client and server workloads in each mTLS interaction/edge, symptom that the certificates are actually exchanged and identities are extracted correctly from them, and moreover all these data about each specific connection and request are collected by the proxies and then stored by Prometheus. On the other hand, figure 7.38 shows how unauthorized connections appear on Kiali: this was the case of a test performed in Chapter 7.1.1 after having applied the workload-to-workload Authorization Policies and trying to make the Traveler service contact the Catalogue one, even if they would not require to interact. Since the identity extracted from the client certificate was not the one of the Ingress Gateway, the request was rejected by the proxy. Instead, connection coming from the Ingress was accepted for obvious reasons.

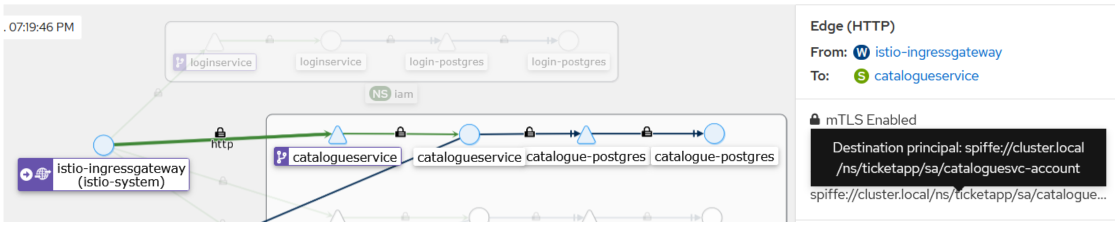

07:19:46 PM
loginservice
loginservice
login-postgres
login-postgres
NS iam
http
istio-ingressgateway (istio-system)
catalogueservice
catalogueservice
catalogue-postgres
catalogue-postgres
Edge (HTTP)
From: istio-ingressgateway
To: catalogueservice
mTLS Enabled
Destination principal: spiffe://cluster.local/ns/ticketapp/sa/cataloguesvc-account
spiffe://cluster.local/ns/ticketapp/sa/catalogue...

Figure 7.37: Kiali Dashboard Visbility 2

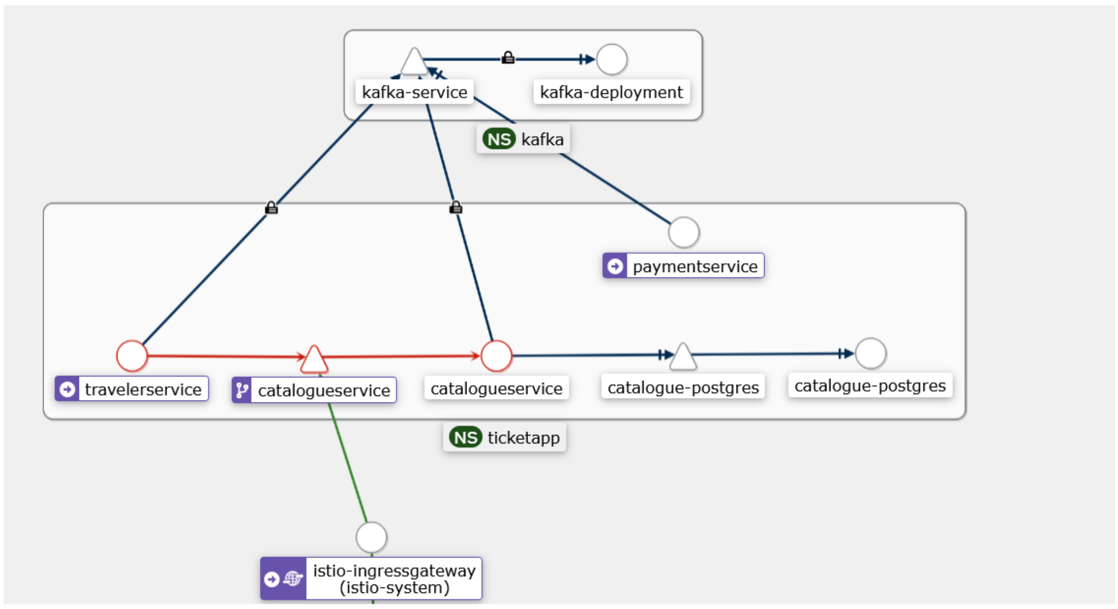
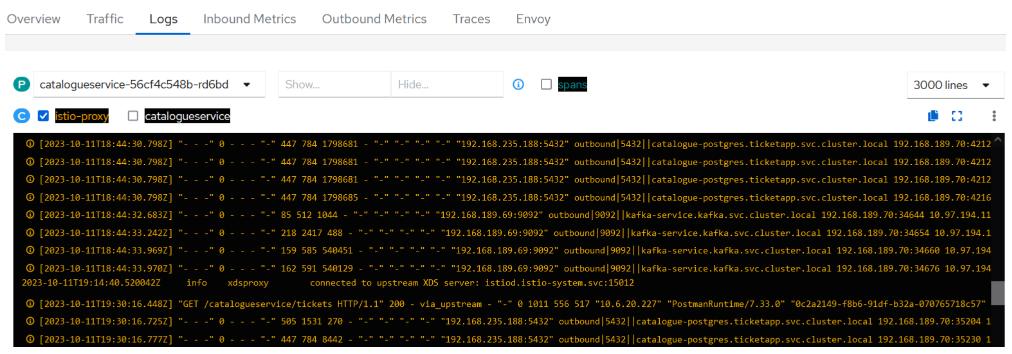


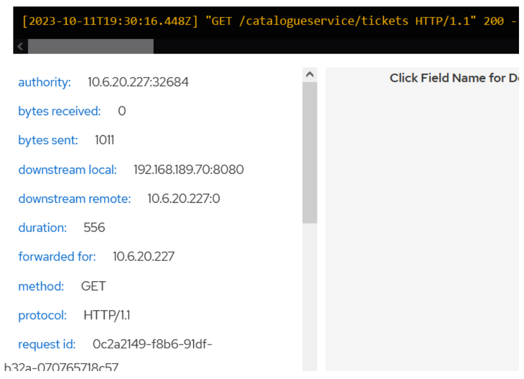
Figure 7.38: Kiali Dashboard Visbility 3

However, all these graphical representations are extracted from the Access Logs and Metrics collected by the proxy of each workload. By clicking on each node/workload, a *Details View* is shown with several information, including inbound and outbound metrics, and specific Envoy instance logs: Figure 7.39 shows the logs referring to the Catalogue service deployment incoming and outgoing connections. For each request log several details are represented, such as whether it is an outgoing or incoming connection, the source and destination IP address and port, as well a timestamp and the duration of the connection. Figures 7.40 and 7.41 show a user-friendly representation of the information contained in each log.



```
[2023-10-11T18:44:30.798Z] "- - -" 0 - - - "-" 447 784 1798681 - "-" "-" "-" "-" "192.168.235.188:5432" outbound|5432||catalogue-postgres.ticketapp.svc.cluster.local 192.168.189.70:4212
[2023-10-11T18:44:30.798Z] "- - -" 0 - - - "-" 447 784 1798681 - "-" "-" "-" "-" "192.168.235.188:5432" outbound|5432||catalogue-postgres.ticketapp.svc.cluster.local 192.168.189.70:4212
[2023-10-11T18:44:30.798Z] "- - -" 0 - - - "-" 447 784 1798681 - "-" "-" "-" "-" "192.168.235.188:5432" outbound|5432||catalogue-postgres.ticketapp.svc.cluster.local 192.168.189.70:4212
[2023-10-11T18:44:30.798Z] "- - -" 0 - - - "-" 447 784 1798685 - "-" "-" "-" "-" "192.168.235.188:5432" outbound|5432||catalogue-postgres.ticketapp.svc.cluster.local 192.168.189.70:4216
[2023-10-11T18:44:32.683Z] "- - -" 0 - - - "-" 85 512 1044 - "-" "-" "-" "-" "192.168.189.69:9092" outbound|9092||kafka-service.kafka.svc.cluster.local 192.168.189.70:34644 10.97.194.11
[2023-10-11T18:44:33.242Z] "- - -" 0 - - - "-" 218 2417 488 - "-" "-" "-" "-" "192.168.189.69:9092" outbound|9092||kafka-service.kafka.svc.cluster.local 192.168.189.70:34654 10.97.194.1
[2023-10-11T18:44:33.969Z] "- - -" 0 - - - "-" 159 585 540451 - "-" "-" "-" "-" "192.168.189.69:9092" outbound|9092||kafka-service.kafka.svc.cluster.local 192.168.189.70:34660 10.97.194
[2023-10-11T18:44:33.970Z] "- - -" 0 - - - "-" 162 591 540129 - "-" "-" "-" "-" "192.168.189.69:9092" outbound|9092||kafka-service.kafka.svc.cluster.local 192.168.189.70:34676 10.97.194
2023-10-11T19:14:40.520042Z     info    xdsproxy        connected to upstream XDS server: istiod.istio-system.svc:15012

[2023-10-11T19:30:16.448Z] "GET /catalogueservice/tickets HTTP/1.1" 200 - via_upstream - "-" 0 1011 556 517 "10.6.20.227" "PostmanRuntime/7.33.0" "0c2a2149-f8b6-91df-b32a-070765718c57"
[2023-10-11T19:30:16.725Z] "- - -" 0 - - - "-" 505 1531 270 - "-" "-" "-" "-" "192.168.235.188:5432" outbound|5432||catalogue-postgres.ticketapp.svc.cluster.local 192.168.189.70:35204 1
[2023-10-11T19:30:16.777Z] "- - -" 0 - - - "-" 447 784 8442 - "-" "-" "-" "-" "192.168.235.188:5432" outbound|5432||catalogue-postgres.ticketapp.svc.cluster.local 192.168.189.70:35230 1
```

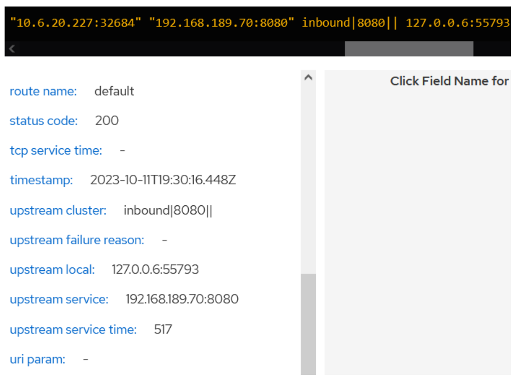
Figure 7.39: Proxy logs of the Catalogue service workload



```
[2023-10-11T19:30:16.448Z] "GET /catalogueservice/tickets HTTP/1.1" 200 -
```

authority:	10.6.20.227:32684
bytes received:	0
bytes sent:	1011
downstream local:	192.168.189.70:8080
downstream remote:	10.6.20.227:0
duration:	556
forwarded for:	10.6.20.227
method:	GET
protocol:	HTTP/1.1
request id:	0c2a2149-f8b6-91df-b32a-070765718c57

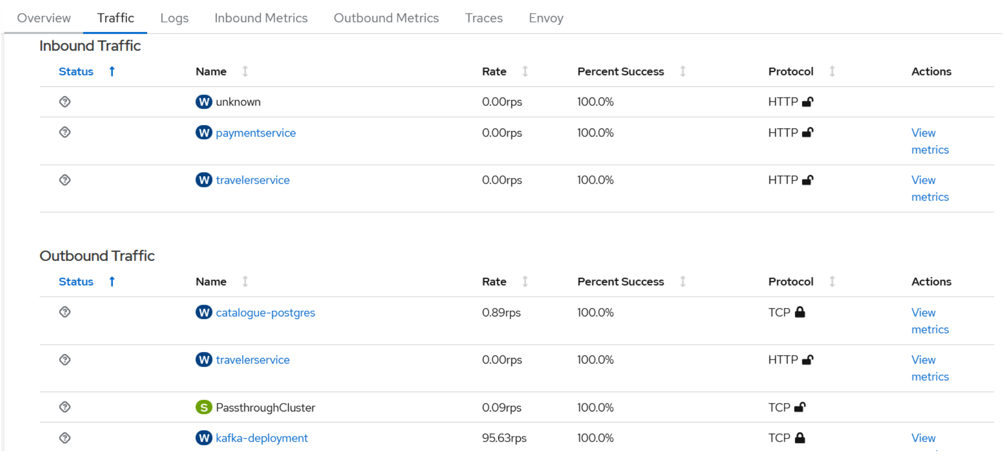
Figure 7.40: First part of the Envoy log



```
"10.6.20.227:32684" "192.168.189.70:8080" inbound|8080|| 127.0.0.6:55793
```

route name:	default
status code:	200
tcp service time:	-
timestamp:	2023-10-11T19:30:16.448Z
upstream cluster:	inbound|8080||
upstream failure reason:	-
upstream local:	127.0.0.6:55793
upstream service:	192.168.189.70:8080
upstream service time:	517
uri param:	-

Figure 7.41: Second part of the Envoy log



Inbound Traffic

Status	Name	Rate	Percent Success	Protocol	Actions
	W unknown	0.00rps	100.0%	HTTP	
	W paymentservice	0.00rps	100.0%	HTTP	View metrics
	W travelerservice	0.00rps	100.0%	HTTP	View metrics

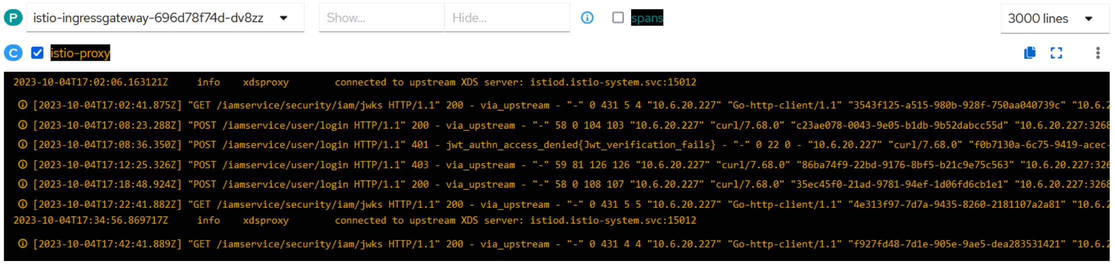
Outbound Traffic

Status	Name	Rate	Percent Success	Protocol	Actions
	W catalogue-postgres	0.89rps	100.0%	TCP	View metrics
	W travelerservice	0.00rps	100.0%	HTTP	View metrics
	S PassthroughCluster	0.09rps	100.0%	TCP	
	W kafka-deployment	95.63rps	100.0%	TCP	View

Figure 7.42: Envoy incoming traffic view

Figure 7.42 also shows a generic view of the incoming traffic for the Catalogue service, which can be further inspected in more details by opening the specific source of interest.

Figure 7.43 shows how also unauthorized and unauthenticated requests are logged by the proxy: in this case it is the traffic received by the ingress gateway, demonstrating that also the latter is able to provide visibility over the incoming traffic.



```
2023-10-04T17:02:06.163121Z     info    xdsproxy        connected to upstream XDS server: istiod.istio-system.svc:15012
 [2023-10-04T17:02:41.875Z] "GET /iamservice/security/iam/jwks HTTP/1.1" 200 - via_upstream - "-" 0 431 5 4 "10.6.20.227" "Go-http-client/1.1" "3543f125-a515-980b-928f-750aa040739c" "10.6.2
 [2023-10-04T17:08:23.288Z] "POST /iamservice/user/login HTTP/1.1" 200 - via_upstream - "-" 58 0 104 103 "10.6.20.227" "curl/7.68.0" "c23ae078-0043-9e05-b1db-9b52dabcc55d" "10.6.20.227:3268
 [2023-10-04T17:08:36.350Z] "POST /iamservice/user/login HTTP/1.1" 401 - jwt_authn_access_denied{Jwt_verification_fails} - "-" 0 22 0 - "10.6.20.227" "curl/7.68.0" "f0b7130a-6c75-9419-acec-
 [2023-10-04T17:12:25.326Z] "POST /iamservice/user/login HTTP/1.1" 403 - via_upstream - "-" 59 81 126 126 "10.6.20.227" "curl/7.68.0" "86ba74f9-22bd-9176-8bf5-b21c9e75c563" "10.6.20.227:326
 [2023-10-04T17:18:48.924Z] "POST /iamservice/user/login HTTP/1.1" 200 - via_upstream - "-" 58 0 108 107 "10.6.20.227" "curl/7.68.0" "35ec45f0-21ad-9781-94ef-1d06fd6cb1e1" "10.6.20.227:3268
 [2023-10-04T17:22:41.882Z] "GET /iamservice/security/iam/jwks HTTP/1.1" 200 - via_upstream - "-" 0 431 5 5 "10.6.20.227" "Go-http-client/1.1" "4e313f97-7d7a-9435-8260-2181107a2a81" "10.6.2
2023-10-04T17:34:56.869717Z     info    xdsproxy        connected to upstream XDS server: istiod.istio-system.svc:15012
 [2023-10-04T17:42:41.889Z] "GET /iamservice/security/iam/jwks HTTP/1.1" 200 - via_upstream - "-" 0 431 4 4 "10.6.20.227" "Go-http-client/1.1" "f927fd48-7d1e-905e-9ae5-dea283531421" "10.6.2
```

Figure 7.43: Logs of unauthorized and unauthenticated requests

It is clear from the pictures above, how Envoy provides detailed logs about HTTP and gRPC requests (as method, path and protocol type and version), while for plain TCP traffic, it only provides generic information like IP addresses, ports, amount of sent/received bytes, K8s services names, etc. Finally, the collected metrics and logs about network traffic and each Envoy status, can be displayed using your own customizable Grafana Dashboard. Figure 7.44 shows one of the possible View obtained by aggregating and elaborating the various Envoy metrics about the workloads within the Mesh. Moreover, also the logs of the request for retrieving the JWKS needed to validate the JWT is present.

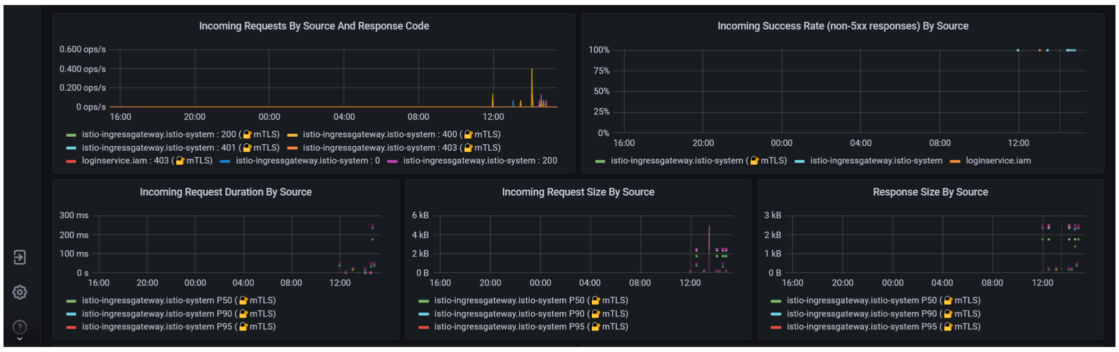


Figure 7.44: Grafana workload-level incoming traffic view

As mentioned in the previous chapters, unfortunately, the Envoy proxy is only able to intercept TCP traffic, thus rendering the authentication, authorisation and visibility that one has over the rest of the traffic, such as UDP or ICMP, practically null and void.

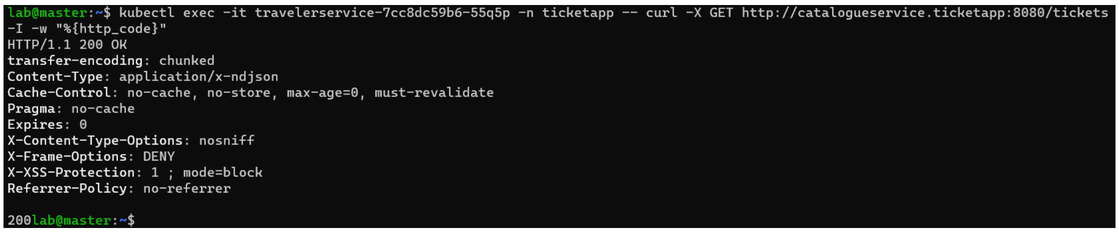
However, it is clear that the traffic observability offered by Istio for TCP and HTTP traffic, perhaps combined with external traffic profiling and analysis tools, enables security-related traffic auditing and monitoring for detection and investigation of network behavior anomalies.

7.2 Palo Alto CN-Series Containerized Firewall

7.2.1 Network Microsegmentation And Traffic Visbility

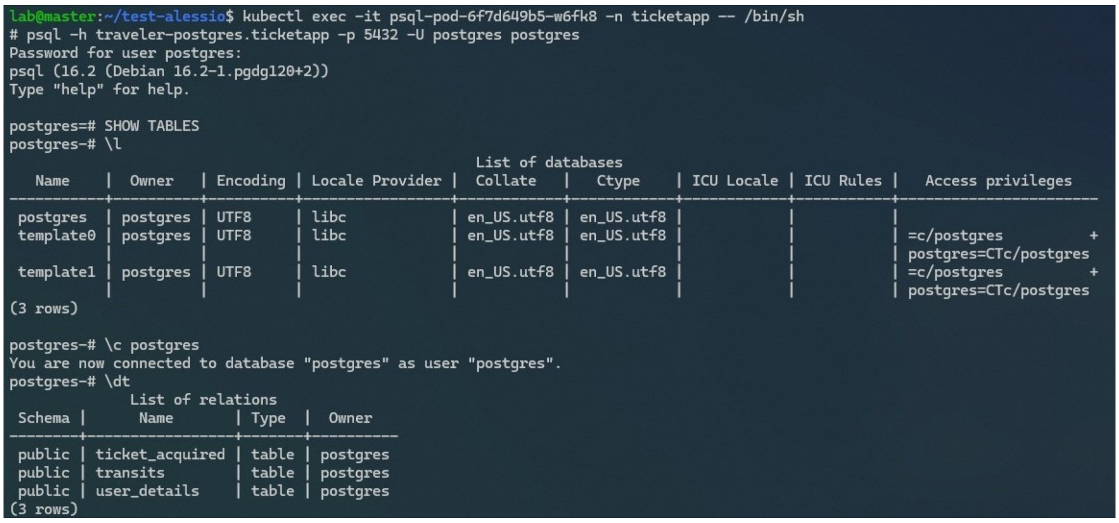
For testing purpose, as for Istio tests, also a pod running a PostgreSQL service (*psql*) was deployed within the ticketapp namespace; the various tests were carried out by simply running *curl*, psql and *ping* tools, and thus generating HTTP, ICMP, Postgres, Kafka, and Zookeper traffic. All the pods within the ticketapp, iam and kafka namespaces were restarted to make the CNI attach their network interface to the firewall pod.

However, prior to the creation of any policy, all types of traffic were permitted. Indeed, Figures 7.45 and 7.46 show how the Traveler service was free to contact the Catalogue endpoint *GET /tickets* and how the psql pod was able to contact the Postgres instance of Traveler, perform the authentication process and then freely access the contents of the Traveler database. These two operations should not occur in a normal situation, but the four microservices should only be contacted by the ingress gateway, while each Postgres instance should only be accessible by the corresponding microservice in charge of handling its sensitive data. We then went on to apply the various deny and allow policies defined in section 6.2.2.



```
lab@master:~$ kubectl exec -it travelerservice-7cc8dc59b6-55q5p -n ticketapp -- curl -X GET http://catalogueservice.ticketapp:8080/tickets
-I -w "%{http_code}"
HTTP/1.1 200 OK
transfer-encoding: chunked
Content-Type: application/x-ndjson
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1 ; mode=block
Referrer-Policy: no-referrer

200lab@master:~$
```

Figure 7.45: Test 1 - Web traffic

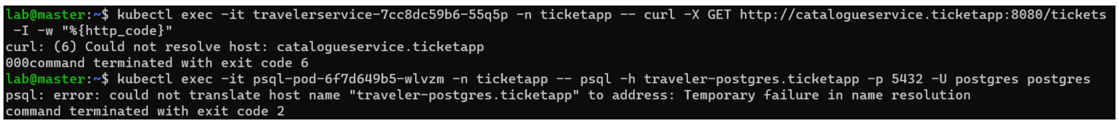


```
lab@master:~/test-alessio$ kubectl exec -it psql-pod-6f7d649b5-w6fk8 -n ticketapp -- /bin/sh
# psql -h traveler-postgres.ticketapp -p 5432 -U postgres postgres
Password for user postgres:
psql (16.2 (Debian 16.2-1.pgdg120+2))
Type "help" for help.

postgres=# SHOW TABLES
postgres-# \l
                                                  List of databases
   Name    |  Owner   | Encoding | Locale Provider |  Collate   |   Ctype    | ICU Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+------------+------------+------------+-----------+-----------------------
 postgres  | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |            |           |
 template0 | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |            |           | =c/postgres          +
           |          |          |                 |            |            |            |           | postgres=CTc/postgres
 template1 | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |            |           | =c/postgres          +
           |          |          |                 |            |            |            |           | postgres=CTc/postgres
(3 rows)

postgres-# \c postgres
You are now connected to database "postgres" as user "postgres".
postgres-# \dt
              List of relations
 Schema |      Name       | Type  |  Owner
--------+-----------------+-------+----------
 public | ticket_acquired | table | postgres
 public | transits        | table | postgres
 public | user_details    | table | postgres
(3 rows)
```

Figure 7.46: Test 2 - Postgres traffic

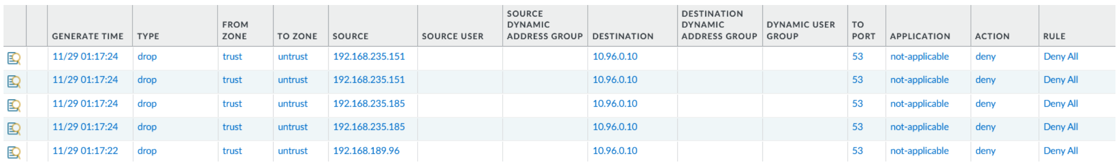
Immediately after the application of the 'Deny All' policy (but still without the 'Allow service

X' policies), no microservice was able to contact DNS and resolve the names of other services, as DNS traffic was also dropped (see Figure 7.47). Figure 7.48 shows the logs of the stopped traffic that corresponded to the Deny All policy, and it is clear that we also have full visibility into the DNS communication intercepted by the firewall, as well as the source and destination IP addresses (10.96.0.10 is the IP address of K8s DNS service). Note that since no DAG was specified in the source and destination of the policies, no DAG is shown in the logs, only the IPs.



```
lab@master:~$ kubectl exec -it travelerservice-7cc8dc59b6-55q5p -n ticketapp -- curl -X GET http://catalogueservice.ticketapp:8080/tickets
 -I -w "%{http_code}"
curl: (6) Could not resolve host: catalogueservice.ticketapp
000command terminated with exit code 6
lab@master:~$ kubectl exec -it psql-pod-6f7d649b5-wlvzm -n ticketapp -- psql -h traveler-postgres.ticketapp -p 5432 -U postgres postgres
psql: error: could not translate host name "traveler-postgres.ticketapp" to address: Temporary failure in name resolution
command terminated with exit code 2
```

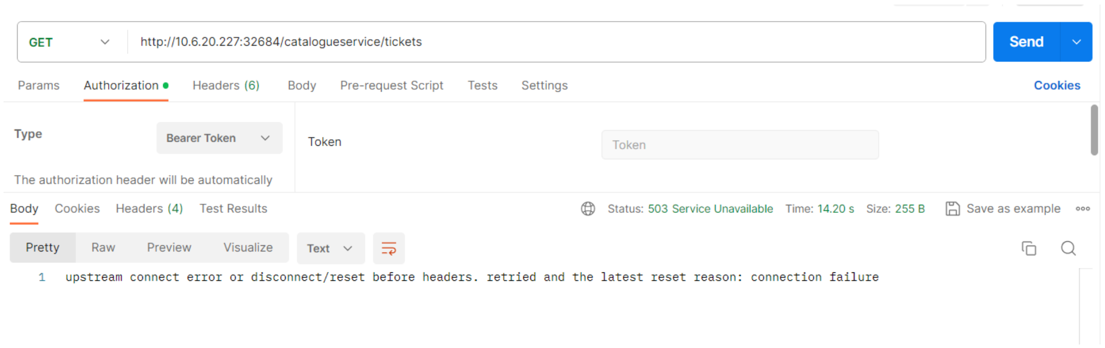
Figure 7.47: Test DNS



		GENERATE TIME	TYPE	FROM ZONE	TO ZONE	SOURCE	SOURCE USER	SOURCE DYNAMIC ADDRESS GROUP	DESTINATION	DESTINATION DYNAMIC ADDRESS GROUP	DYNAMIC USER GROUP	TO PORT	APPLICATION	ACTION	RULE
		11/29 01:17:24	drop	trust	untrust	192.168.235.151			10.96.0.10			53	not-applicable	deny	Deny All
		11/29 01:17:24	drop	trust	untrust	192.168.235.151			10.96.0.10			53	not-applicable	deny	Deny All
		11/29 01:17:24	drop	trust	untrust	192.168.235.185			10.96.0.10			53	not-applicable	deny	Deny All
		11/29 01:17:24	drop	trust	untrust	192.168.235.185			10.96.0.10			53	not-applicable	deny	Deny All
		11/29 01:17:22	drop	trust	untrust	192.168.189.96			10.96.0.10			53	not-applicable	deny	Deny All

Figure 7.48: Logs DNS traffic dropped

After ensuring that DNS traffic was allowed within the cluster, another test was performed to confirm that none of the microservices within the cluster could also be contacted by an external entity, and thus that the firewall at that time also applied the deny all policy on incoming traffic, i.e. that coming from the ingress gateway. The gateway was still configured to act as a reverse proxy and route incoming HTTTP requests to the specific microservice (i.e. its specific clusterIP) using the prefix contained in the request URL. The requests sent from my laptop to the incoming gateway address in Figures 7.49 and 7.50, show how the traffic actually reached the gateway (since the response was provided by an 'istio envoy' server), was then forwarded to the service pod (since its configuration worked fine before applying the policies), but before reaching the specific endpoint, the firewall in front of the service pod dropped the packets and communication failed after no response was provided within a specified time period. Note that the firewall was not connected to the inbound gateway traffic, but only to the ticketing application's pods.



GET http://10.6.20.227:32684/catalogueservice/tickets

Type: Bearer Token

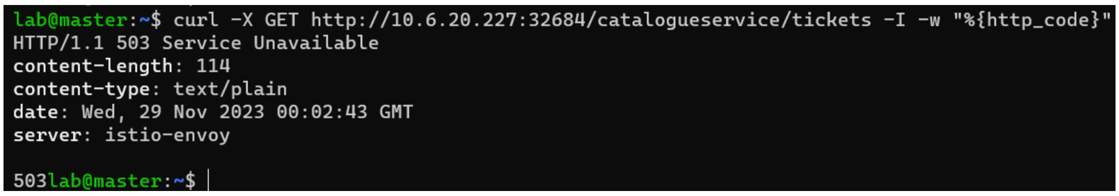
Token

The authorization header will be automatically

Status: 503 Service Unavailable Time: 14.20 s Size: 255 B

```
upstream connect error or disconnect/reset before headers. retried and the latest reset reason: connection failure
```

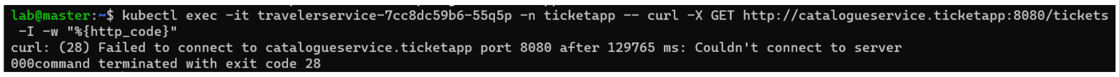
Figure 7.49: Test ingress traffic with Postman



```
lab@master:~$ curl -X GET http://10.6.20.227:32684/catalogueservice/tickets -I -w "%{http_code}"
HTTP/1.1 503 Service Unavailable
content-length: 114
content-type: text/plain
date: Wed, 29 Nov 2023 00:02:43 GMT
server: istio-envoy

503lab@master:~$
```

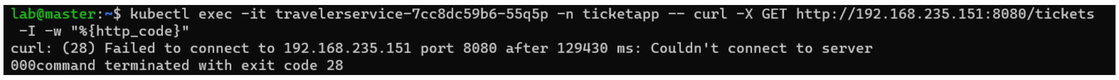
Figure 7.50: Test ingress traffic with curl

The various ALLOW policies defined for the various services, including DNS, were then applied. Repeating the same tests as before, the pod of the Traveler service was still unable to contact the Catalogue service, either using its DNS name (and thus the IP address of the Catalogue cluster) or the actual IP address of the pod (see Figures 7.51 and 7.52). However, this time the test showed firstly that DNS communication was now allowed, as DNS resolution was successful and curl did not show any related errors, as had happened previously in Figure 7.47, and consequently that the "Allow DNS" policy worked correctly. Secondly, the deny policy worked correctly for both traffic using the address of the clusterIP service and the IP of the pod as the destination of the packets. As can be seen, the connection failed due to a timeout: this occurred because the action taken by the deny all policy (based on my configuration) was simply to drop packets rather than reset the connection. Figure 7.53 shows the log of the allowed DNS traffic requested by the Traveler service for DNS resolution and the subsequent HTTP traffic to the clusterIP service resolved and denied by the firewall.



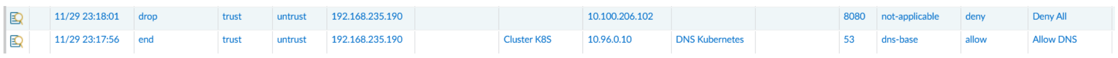
```
lab@master:~$ kubectl exec -it travelerservice-7cc8dc59b6-55q5p -n ticketapp -- curl -X GET http://catalogueservice.ticketapp:8080/tickets
 -I -w "%{http_code}"
curl: (28) Failed to connect to catalogueservice.ticketapp port 8080 after 129765 ms: Couldn't connect to server
000command terminated with exit code 28
```

Figure 7.51: Test 3 - Web traffic to the clusterIP address after policies enforcing



```
lab@master:~$ kubectl exec -it travelerservice-7cc8dc59b6-55q5p -n ticketapp -- curl -X GET http://192.168.235.151:8080/tickets
 -I -w "%{http_code}"
curl: (28) Failed to connect to 192.168.235.151 port 8080 after 129430 ms: Couldn't connect to server
000command terminated with exit code 28
```

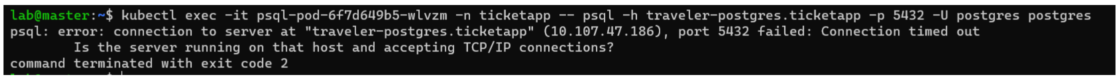
Figure 7.52: Test 4 - Web traffic to the pod IP address after policies enforcing



11/29 23:18:01	drop	trust	untrust	192.168.235.190			10.100.206.102			8080	not-applicable	deny	Deny All
11/29 23:17:56	end	trust	untrust	192.168.235.190		Cluster K8S	10.96.0.10	DNS Kubernetes		53	dns-base	allow	Allow DNS

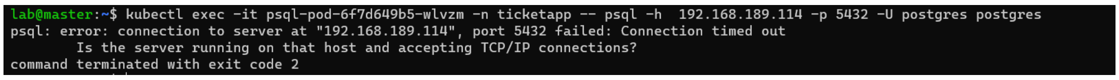
Figure 7.53: Logs about DNS and HTTP traffic leaving the Traveler service pod

The psql pod could also contact DNS, but was now unable to contact the Postgres service, either using its DNS name or the actual IP address of the pod (see Figures 7.54 and 7.55). Figure 7.56 shows the logs corresponding to the test in Figure 7.54.



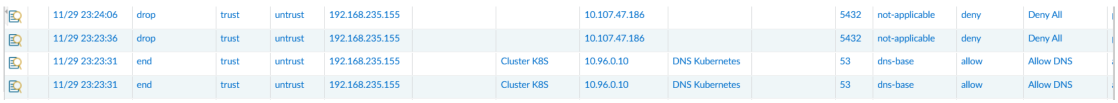
```
lab@master:~$ kubectl exec -it psql-pod-6f7d649b5-wlvzm -n ticketapp -- psql -h traveler-postgres.ticketapp -p 5432 -U postgres postgres
psql: error: connection to server at "traveler-postgres.ticketapp" (10.107.47.186), port 5432 failed: Connection timed out
        Is the server running on that host and accepting TCP/IP connections?
command terminated with exit code 2
```

Figure 7.54: Test 5 - Postgres traffic to the cluster IP address after policies enforcing



```
lab@master:~$ kubectl exec -it psql-pod-6f7d649b5-wlvzm -n ticketapp -- psql -h  192.168.189.114 -p 5432 -U postgres postgres
psql: error: connection to server at "192.168.189.114", port 5432 failed: Connection timed out
        Is the server running on that host and accepting TCP/IP connections?
command terminated with exit code 2
```

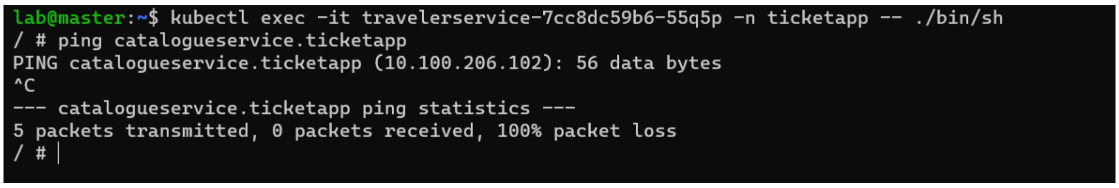
Figure 7.55: Test 6 - Postgres traffic to the pod IP address after policies enforcing



11/29 23:24:06	drop	trust	untrust	192.168.235.155			10.107.47.186			5432	not-applicable	deny	Deny All
11/29 23:23:36	drop	trust	untrust	192.168.235.155			10.107.47.186			5432	not-applicable	deny	Deny All
11/29 23:23:31	end	trust	untrust	192.168.235.155		Cluster K8S	10.96.0.10	DNS Kubernetes		53	dns-base	allow	Allow DNS
11/29 23:23:31	end	trust	untrust	192.168.235.155		Cluster K8S	10.96.0.10	DNS Kubernetes		53	dns-base	allow	Allow DNS

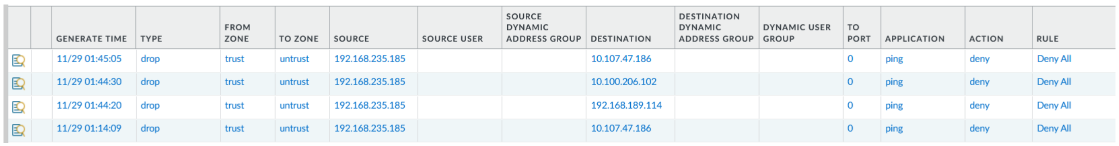
Figure 7.56: Logs about DNS and Postgres traffic leaving the Psql service pod

The same was true for some ping tests performed towards various microservices: the ICMP traffic was dropped by the firewall (Figure 7.57 shows the attempt, while Figure 7.58 shows the corresponding logs).



```
lab@master:~$ kubectl exec -it travelerservice-7cc8dc59b6-55q5p -n ticketapp -- ./bin/sh
/ # ping catalogueservice.ticketapp
PING catalogueservice.ticketapp (10.100.206.102): 56 data bytes
^C
--- catalogueservice.ticketapp ping statistics ---
5 packets transmitted, 0 packets received, 100% packet loss
/ # |
```

Figure 7.57: Test ICMP traffic after policies enforcing

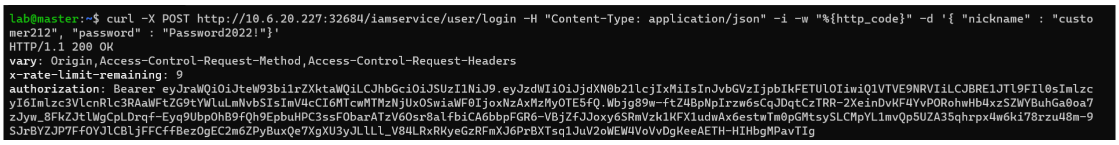


GENERATE TIME	TYPE	FROM ZONE	TO ZONE	SOURCE	SOURCE USER	SOURCE DYNAMIC ADDRESS GROUP	DESTINATION	DESTINATION DYNAMIC ADDRESS GROUP	DYNAMIC USER GROUP	TO PORT	APPLICATION	ACTION	RULE
11/29 01:45:05	drop	trust	untrust	192.168.235.185			10.107.47.186			0	ping	deny	Deny All
11/29 01:44:30	drop	trust	untrust	192.168.235.185			10.100.206.102			0	ping	deny	Deny All
11/29 01:44:20	drop	trust	untrust	192.168.235.185			192.168.189.114			0	ping	deny	Deny All
11/29 01:14:09	drop	trust	untrust	192.168.235.185			10.107.47.186			0	ping	deny	Deny All

Figure 7.58: Logs of ICMP traffic dropped by the firewall

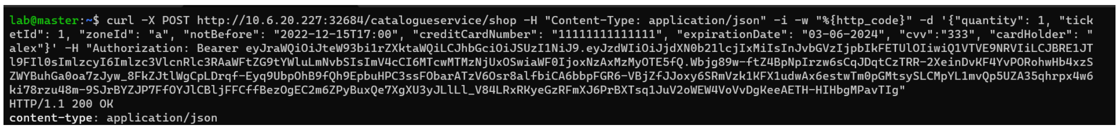
After making sure that the Deny All policy was blocking all unwanted traffic, I had to check whether the 'Allow Service X to Service Y' policy was working properly, and thus that all service pairs (and only them, since all other traffic was blocked as seen above) that needed to communicate were able to do so.

I then ran a few requests from my laptop (i.e. from outside the cluster) to check that everything was working properly. So, I first logged in by contacting the Login microservice through the Istio ingress gateway (Figure 7.59) and then, using the returned JWT, I sent a request to the *POST /shop* endpoint of the Catalogue service (Figure 7.60: a valid response was returned, with both microservices, so the policy allowing HTTP traffic between the ingress gateway and the various microservices worked correctly. Subsequently, by also contacting the *GET /my/tickets/* endpoint of the Traveler service (Figure 7.61) and the *GET /transactions/* endpoint of the Payment service (Figure 7.62), it became clear that the last record written by the two microservices within their Postgres instance referred to the same ticket purchase request that had been sent to Catalogue shortly before, and therefore that the three microservices were communicating correctly. This implicitly demonstrates that communication between the three microservices and Kafka was allowed to take place correctly (since communication between them was designed to take place only through the message broker), as was communication between Kafka and Zookeeper (otherwise communication would have failed). Finally, of course, it was also shown that each microservice was able to contact its own instance of Postgres to store and retrieve data.



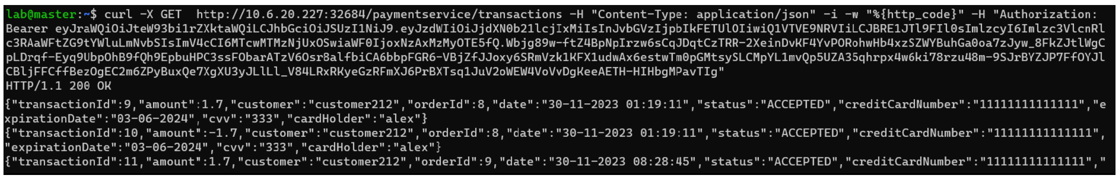
```
lab@master:~$ curl -X POST http://10.6.20.227:32684/iamservice/user/login -H "Content-Type: application/json" -i -w "%{http_code}" -d '{ "nickname" : "custo
mer212", "password" : "Password2022!"}'
HTTP/1.1 200 OK
vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
x-rate-limit-remaining: 9
authorization: Bearer eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxMiIsInJvbGVzIjpbIkFETUlOIiwiQ1VTVE9NRVIiLCJBRE1JTl9FIl0sImlzc
yI6Imlzc3VlcnRlc3RAaWFtZG9tYWluLmNvbSIsImV4cCI6MTcwMTMzNjUxOSwiaWF0IjoxNzAxMzMyOTE5fQ.Wbjg89w-ftZ4BpNpIrzw6sCqJDqtCzTRR-2XeinDvKF4YvPORohwHb4xzSZWYBuhGa0oa7
zJyw_8FkZJtlWgCpLDrqf-Eyq9UbpOhB9fQh9EpbuHPC3ssFObarATzV6Osr8alfbiCA6bbpFGR6-VBjZfJJoxy6SRmVzk1KFX1udwAx6estwTm0pGMtsySLCMpYL1mvQp5UZA35qhrpx4w6ki78rzu48m-9
SJrBYZJP7FfOYJlCBljFFCffBezOgEC2m6ZPyBuxQe7XgXU3yJLlLl_V84LRxRKyeGzRFmXJ6PrBXTsq1JuV2oWEW4VoVvDgKeeAETH-HIHbgMPavTIg
```

Figure 7.59: Test HTTP request to Login service



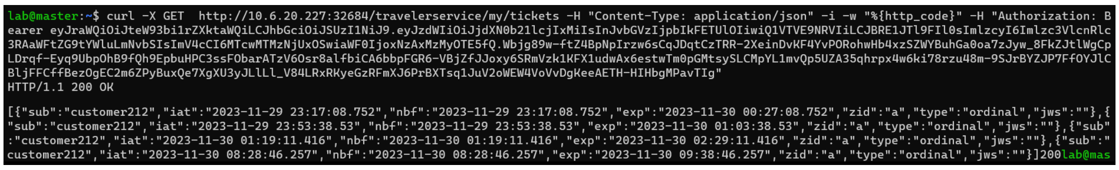
```
lab@master:~$ curl -X POST http://10.6.20.227:32684/catalogueservice/shop -H "Content-Type: application/json" -i -w "%{http_code}" -d '{"quantity": 1, "tick
etId": 1, "zoneId": "a", "notBefore": "2022-12-15T17:00", "creditCardNumber": "11111111111111", "expirationDate": "03-06-2024", "cvv":"333", "cardHolder": "
alex"}' -H "Authorization: Bearer eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxMiIsInJvbGVzIjpbIkFETUlOIiwiQ1VTVE9NRVIiLCJBRE1JT
l9FIl0sImlzcyI6Imlzc3VlcnRlc3RAaWFtZG9tYWluLmNvbSIsImV4cCI6MTcwMTMzNjUxOSwiaWF0IjoxNzAxMzMyOTE5fQ.Wbjg89w-ftZ4BpNpIrzw6sCqJDqtCzTRR-2XeinDvKF4YvPORohwHb4xzS
ZWYBuhGa0oa7zJyw_8FkZJtlWgCpLDrqf-Eyq9UbpOhB9fQh9EpbuHPC3ssFObarATzV6Osr8alfbiCA6bbpFGR6-VBjZfJJoxy6SRmVzk1KFX1udwAx6estwTm0pGMtsySLCMpYL1mvQp5UZA35qhrpx4w6
ki78rzu48m-9SJrBYZJP7FfOYJlCBljFFCffBezOgEC2m6ZPyBuxQe7XgXU3yJLlLl_V84LRxRKyeGzRFmXJ6PrBXTsq1JuV2oWEW4VoVvDgKeeAETH-HIHbgMPavTIg"
HTTP/1.1 200 OK
content-type: application/json
```

Figure 7.60: Test HTTP request to Catalogue service



```
lab@master:~$ curl -X GET  http://10.6.20.227:32684/paymentservice/transactions -H "Content-Type: application/json" -i -w "%{http_code}" -H "Authorization:
Bearer eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxMiIsInJvbGVzIjpbIkFETUlOIiwiQ1VTVE9NRVIiLCJBRE1JTl9FIl0sImlzcyI6Imlzc3VlcnRl
c3RAaWFtZG9tYWluLmNvbSIsImV4cCI6MTcwMTMzNjUxOSwiaWF0IjoxNzAxMzMyOTE5fQ.Wbjg89w-ftZ4BpNpIrzw6sCqJDqtCzTRR-2XeinDvKF4YvPORohwHb4xzSZWYBuhGa0oa7zJyw_8FkZJtlWgC
pLDrqf-Eyq9UbpOhB9fQh9EpbuHPC3ssFObarATzV6Osr8alfbiCA6bbpFGR6-VBjZfJJoxy6SRmVzk1KFX1udwAx6estwTm0pGMtsySLCMpYL1mvQp5UZA35qhrpx4w6ki78rzu48m-9SJrBYZJP7FfOYJl
CBljFFCffBezOgEC2m6ZPyBuxQe7XgXU3yJLlLl_V84LRxRKyeGzRFmXJ6PrBXTsq1JuV2oWEW4VoVvDgKeeAETH-HIHbgMPavTIg"
HTTP/1.1 200 OK
{"transactionId":9,"amount":1.7,"customer":"customer212","orderId":8,"date":"30-11-2023 01:19:11","status":"ACCEPTED","creditCardNumber":"11111111111111","e
xpirationDate":"03-06-2024","cvv":"333","cardHolder":"alex"}
{"transactionId":10,"amount":-1.7,"customer":"customer212","orderId":8,"date":"30-11-2023 01:19:11","status":"ACCEPTED","creditCardNumber":"11111111111111",
"expirationDate":"03-06-2024","cvv":"333","cardHolder":"alex"}
{"transactionId":11,"amount":1.7,"customer":"customer212","orderId":9,"date":"30-11-2023 08:28:45","status":"ACCEPTED","creditCardNumber":"11111111111111","
```

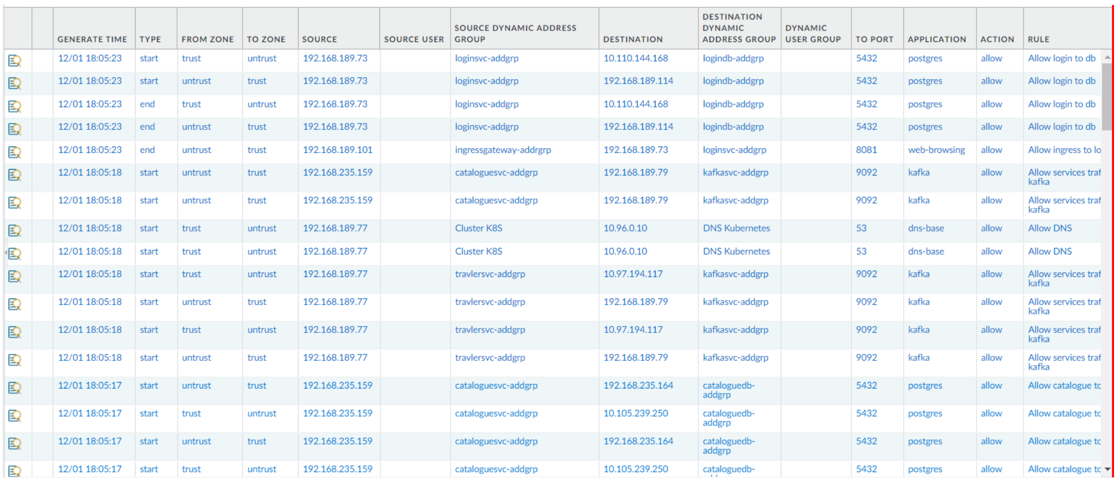
Figure 7.61: Test HTTP request to Payment service



```
lab@master:~$ curl -X GET  http://10.6.20.227:32684/travelerservice/my/tickets -H "Content-Type: application/json" -i -w "%{http_code}" -H "Authorization: B
earer eyJraWQiOiJteW93bi1rZXktaWQiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJjdXN0b21lcjIxMiIsInJvbGVzIjpbIkFETUlOIiwiQ1VTVE9NRVIiLCJBRE1JTl9FIl0sImlzcyI6Imlzc3VlcnRlc
3RAaWFtZG9tYWluLmNvbSIsImV4cCI6MTcwMTMzNjUxOSwiaWF0IjoxNzAxMzMyOTE5fQ.Wbjg89w-ftZ4BpNpIrzw6sCqJDqtCzTRR-2XeinDvKF4YvPORohwHb4xzSZWYBuhGa0oa7zJyw_8FkZJtlWgCp
LDrqf-Eyq9UbpOhB9fQh9EpbuHPC3ssFObarATzV6Osr8alfbiCA6bbpFGR6-VBjZfJJoxy6SRmVzk1KFX1udwAx6estwTm0pGMtsySLCMpYL1mvQp5UZA35qhrpx4w6ki78rzu48m-9SJrBYZJP7FfOYJlC
BljFFCffBezOgEC2m6ZPyBuxQe7XgXU3yJLlLl_V84LRxRKyeGzRFmXJ6PrBXTsq1JuV2oWEW4VoVvDgKeeAETH-HIHbgMPavTIg"
HTTP/1.1 200 OK

[{"sub":"customer212","iat":"2023-11-29 23:17:08.752","nbf":"2023-11-29 23:17:08.752","exp":"2023-11-30 00:27:08.752","zid":"a","type":"ordinal","jws":""},{
"sub":"customer212","iat":"2023-11-29 23:53:38.53","nbf":"2023-11-29 23:53:38.53","exp":"2023-11-30 01:03:38.53","zid":"a","type":"ordinal","jws":""},{"sub"
:"customer212","iat":"2023-11-30 01:19:11.416","nbf":"2023-11-30 01:19:11.416","exp":"2023-11-30 02:29:11.416","zid":"a","type":"ordinal","jws":""},{"sub":"
customer212","iat":"2023-11-30 08:28:46.257","nbf":"2023-11-30 08:28:46.257","exp":"2023-11-30 09:38:46.257","zid":"a","type":"ordinal","jws":""}]200lab@mas
```

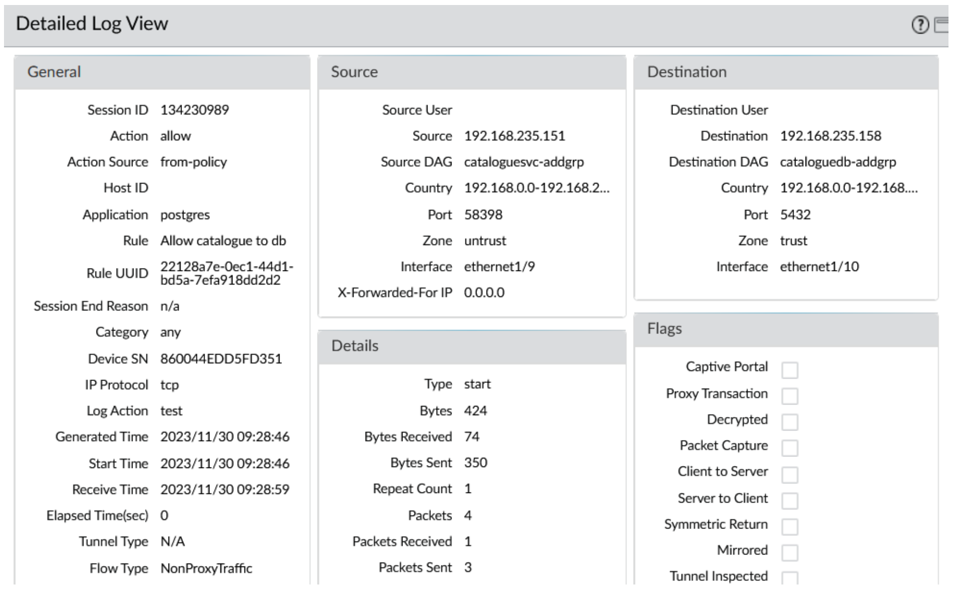
Figure 7.62: Test HTTP request to Traveler service

However, explicit proof was provided by the logs inspected by Panorama. Figure 7.63 shows that the various authorisation policies matched and worked correctly for the services within the cluster, and also demonstrated and highlighted that with the CN series we have full visibility of any traffic received by the firewall protected pods, as the CN series is able to provide full L7 visibility. Figure 7.64 shows some of the detailed information that can be obtained for each specific traffic log.



GENERATE TIME	TYPE	FROM ZONE	TO ZONE	SOURCE	SOURCE USER	SOURCE DYNAMIC ADDRESS GROUP	DESTINATION	DESTINATION DYNAMIC ADDRESS GROUP	DYNAMIC USER GROUP	TO PORT	APPLICATION	ACTION	RULE
12/01 18:05:23	start	trust	untrust	192.168.189.73		loginsvc-addgrp	10.110.144.168	logindb-addgrp		5432	postgres	allow	Allow login to db
12/01 18:05:23	start	untrust	trust	192.168.189.73		loginsvc-addgrp	192.168.189.114	logindb-addgrp		5432	postgres	allow	Allow login to db
12/01 18:05:23	end	trust	untrust	192.168.189.73		loginsvc-addgrp	10.110.144.168	logindb-addgrp		5432	postgres	allow	Allow login to db
12/01 18:05:23	end	untrust	trust	192.168.189.73		loginsvc-addgrp	192.168.189.114	logindb-addgrp		5432	postgres	allow	Allow login to db
12/01 18:05:23	end	untrust	trust	192.168.189.101		ingressgateway-addrgrp	192.168.189.73	loginsvc-addgrp		8081	web-browsing	allow	Allow ingress to lo
12/01 18:05:18	start	untrust	trust	192.168.235.159		cataloguesvc-addgrp	192.168.189.79	kafkasvc-addgrp		9092	kafka	allow	Allow services traf kafka
12/01 18:05:18	start	untrust	trust	192.168.235.159		cataloguesvc-addgrp	192.168.189.79	kafkasvc-addgrp		9092	kafka	allow	Allow services traf kafka
12/01 18:05:18	start	trust	untrust	192.168.189.77		Cluster K8S	10.96.0.10	DNS Kubernetes		53	dns-base	allow	Allow DNS
12/01 18:05:18	start	trust	untrust	192.168.189.77		Cluster K8S	10.96.0.10	DNS Kubernetes		53	dns-base	allow	Allow DNS
12/01 18:05:18	start	trust	untrust	192.168.189.77		travlersvc-addgrp	10.97.194.117	kafkasvc-addgrp		9092	kafka	allow	Allow services traf kafka
12/01 18:05:18	start	untrust	trust	192.168.189.77		travlersvc-addgrp	192.168.189.79	kafkasvc-addgrp		9092	kafka	allow	Allow services traf kafka
12/01 18:05:18	start	trust	untrust	192.168.189.77		travlersvc-addgrp	10.97.194.117	kafkasvc-addgrp		9092	kafka	allow	Allow services traf kafka
12/01 18:05:18	start	untrust	trust	192.168.189.77		travlersvc-addgrp	192.168.189.79	kafkasvc-addgrp		9092	kafka	allow	Allow services traf kafka
12/01 18:05:17	start	untrust	trust	192.168.235.159		cataloguesvc-addgrp	192.168.235.164	cataloguedb-addgrp		5432	postgres	allow	Allow catalogue tc
12/01 18:05:17	start	trust	untrust	192.168.235.159		cataloguesvc-addgrp	10.105.239.250	cataloguedb-addgrp		5432	postgres	allow	Allow catalogue tc
12/01 18:05:17	start	untrust	trust	192.168.235.159		cataloguesvc-addgrp	192.168.235.164	cataloguedb-addgrp		5432	postgres	allow	Allow catalogue tc
12/01 18:05:17	start	trust	untrust	192.168.235.159		cataloguesvc-addgrp	10.105.239.250	cataloguedb-		5432	postgres	allow	Allow catalogue tc

Figure 7.63: Logs about all the services traffic intercepted by the firewall



Detailed Log View

General

Session ID	134230989
Action	allow
Action Source	from-policy
Host ID	
Application	postgres
Rule	Allow catalogue to db
Rule UUID	22128a7e-0ec1-44d1-bd5a-7efa918dd2d2
Session End Reason	n/a
Category	any
Device SN	860044EDD5FD351
IP Protocol	tcp
Log Action	test
Generated Time	2023/11/30 09:28:46
Start Time	2023/11/30 09:28:46
Receive Time	2023/11/30 09:28:59
Elapsed Time(sec)	0
Tunnel Type	N/A
Flow Type	NonProxyTraffic

Source

Source User	
Source	192.168.235.151
Source DAG	cataloguesvc-addgrp
Country	192.168.0.0-192.168.2...
Port	58398
Zone	untrust
Interface	ethernet1/9
X-Forwarded-For IP	0.0.0.0

Details

Type	start
Bytes	424
Bytes Received	74
Bytes Sent	350
Repeat Count	1
Packets	4
Packets Received	1
Packets Sent	3

Destination

Destination User	
Destination	192.168.235.158
Destination DAG	cataloguedb-addgrp
Country	192.168.0.0-192.168....
Port	5432
Zone	trust
Interface	ethernet1/10

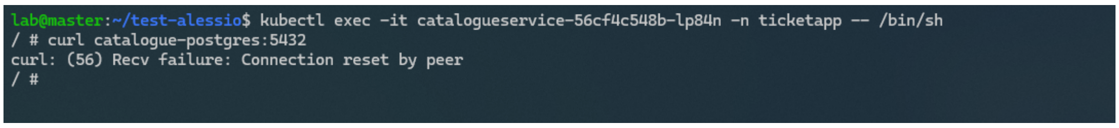
Flags

Captive Portal	☐
Proxy Transaction	☐
Decrypted	☐
Packet Capture	☐
Client to Server	☐
Server to Client	☐
Symmetric Return	☐
Mirrored	☐
Tunnel Inspected	☐

Figure 7.64: Detailed view of a log

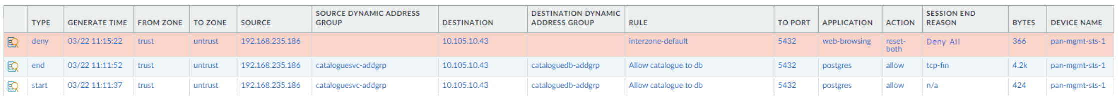
It should be noted that for each 'allow policy', not only the source and destination DAGs and the service port were specified, but also the specific type of application running on each port. In Figure 7.65, a test was performed, letting the Catalogue service contact the pod running its Postgres instance on the same Postgres port, but using an HTTP request. Although this situation did not make sense, the test (and the logs in Figure 7.66) showed that the firewall does indeed have full L7 visibility and is able to inspect and recognise the application content of the packets, detecting that even though the traffic between the two pods was on a correct service port, it was actually HTTP traffic rather than Postgres traffic, and was therefore denied. This demonstrated that the CN series can be used to achieve fine-grained micro-segmentation, based not only on the

IPs of the source and destination pods, but also on the specific application traffic (and service types) exchanged between the two entities.



```
lab@master:~/test-alessio$ kubectl exec -it catalogueservice-56cf4c548b-lp84n -n ticketapp -- /bin/sh
/ # curl catalogue-postgres:5432
curl: (56) Recv failure: Connection reset by peer
/ #
```

Figure 7.65: Test HTTP traffic towards the Postgres service



	TYPE	GENERATE TIME	FROM ZONE	TO ZONE	SOURCE	SOURCE DYNAMIC ADDRESS GROUP	DESTINATION	DESTINATION DYNAMIC ADDRESS GROUP	RULE	TO PORT	APPLICATION	ACTION	SESSION END REASON	BYTES	DEVICE NAME
	deny	03/22 11:15:22	trust	untrust	192.168.235.186		10.105.10.43		interzone-default	5432	web-browsing	reset-both	Deny All	366	pan-mgmt-sts-1
	end	03/22 11:11:52	trust	untrust	192.168.235.186	cataloguesvc-addgrp	10.105.10.43	cataloguedb-addgrp	Allow catalogue to db	5432	postgres	allow	tcp-fin	4.2k	pan-mgmt-sts-1
	start	03/22 11:11:37	trust	untrust	192.168.235.186	cataloguesvc-addgrp	10.105.10.43	cataloguedb-addgrp	Allow catalogue to db	5432	postgres	allow	n/a	424	pan-mgmt-sts-1

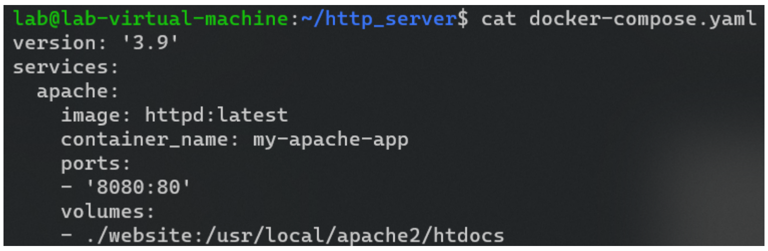
Figure 7.66: Logs about the HTTP traffic denied towards the Postgres service

However, these logs generated by the firewall and collected by Panorama can optionally be integrated with a monitoring and alerting configuration, to report any abnormal behaviour or security incidents detected by the firewall by leveraging the defined security policies. Finally, it is clear that complete visibility into all network activity is provided and that almost complete micro-segmentation has been achieved through CN-series: in fact, unlike Istio, CN-series alone does not automatically provide any strong identity for our Kubernetes services to authenticate each other. However, if a valid TLS certificate is somehow provided to the various pods, a TLS client authentication mechanism can be configured on each firewall (also generating a custom server-side certificate to be deployed within each firewall acting as a TLS server). Through a client authorisation list, the firewall can check the Subject or Subject Alt Name of the client certificate, and if it does not match an identifier in the authorisation list, authentication is denied. This, combined with the specific L7 authorisation policies defined for our tests, and mTLS adoption, will achieve zero trust within our network. Unfortunately, at the same time, this would require us to have our own Certificate Authority embedded in a valid PKI, to manually configure and manage the issuance of valid X.509 certificates for each workload within the cluster, and to properly set up the pods to establish mTLS communication each time they wish to contact the server, ensuring that only outgoing mTLS connections are used during communication.

7.2.2 URL filtering And Advanced Threat Prevention

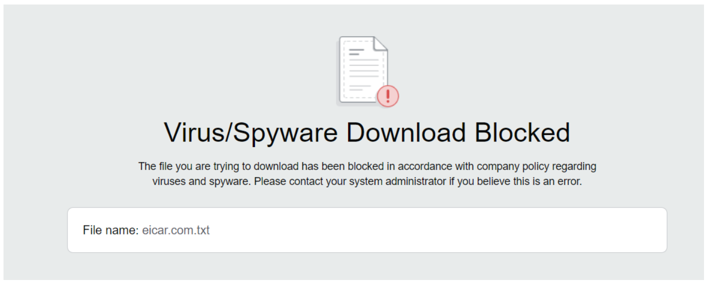
With regard to advanced threat prevention functionality, the antivirus profile set for our security policy was tested using a simple EICAR file. The EICAR antivirus test file is a computer file developed by the European Institute for Computer Antivirus Research (EICAR) and the Computer Antivirus Research Organisation (CARO) to test the response of computer antivirus programmes. Instead of using real malware, which could cause real damage, this test file allows antivirus software to be tested without having to use a real computer virus.

To simulate a lateral movement scenario, an HTTP server was then deployed within our Kubernetes cluster as a pod (Figure 7.67), which hosts the EICAR file downloaded from the official site [EICAR] , and an allow policy which permitted HTTP traffic between the client pod and the server hosting the file was set, also enforcing the antivirus profile configured in section 6.2.3. Subsequently, an attempt was made to download the hosted file by sending an appropriate HTTP request to the server: the connection matched the authorisation policy, but the virus present in the traffic was detected and the firewall immediately interrupted the connection, providing as a response the warning HTML page displayed in Figure 7.68. Naturally, logs were generated and archived in Panorama, highlighting the threat and thus the reason why the connection was interrupted (Figure 7.69.



```
lab@lab-virtual-machine:~/http_server$ cat docker-compose.yaml
version: '3.9'
services:
  apache:
    image: httpd:latest
    container_name: my-apache-app
    ports:
    - '8080:80'
    volumes:
    - ./website:/usr/local/apache2/htdocs
```

Figure 7.67: YAML file of the HTTP server hosting the EICAR file

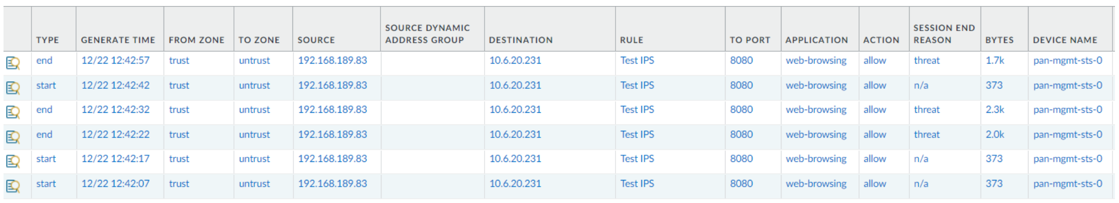


Virus/Spyware Download Blocked

The file you are trying to download has been blocked in accordance with company policy regarding viruses and spyware. Please contact your system administrator if you believe this is an error.

File name: eicar.com.txt

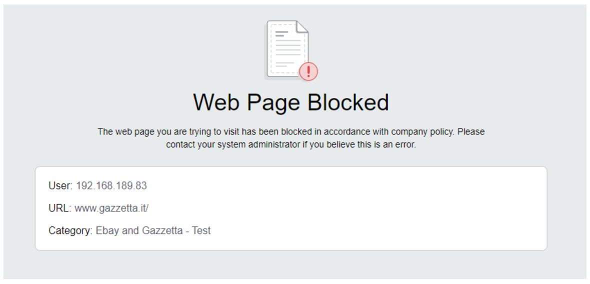
Figure 7.68: Warning HTML page returned by the firewall - 1



	TYPE	GENERATE TIME	FROM ZONE	TO ZONE	SOURCE	SOURCE DYNAMIC ADDRESS GROUP	DESTINATION	RULE	TO PORT	APPLICATION	ACTION	SESSION END REASON	BYTES	DEVICE NAME
	end	12/22 12:42:57	trust	untrust	192.168.189.83		10.6.20.231	Test IPS	8080	web-browsing	allow	threat	1.7k	pan-mgmt-sts-0
	start	12/22 12:42:42	trust	untrust	192.168.189.83		10.6.20.231	Test IPS	8080	web-browsing	allow	n/a	373	pan-mgmt-sts-0
	end	12/22 12:42:32	trust	untrust	192.168.189.83		10.6.20.231	Test IPS	8080	web-browsing	allow	threat	2.3k	pan-mgmt-sts-0
	end	12/22 12:42:22	trust	untrust	192.168.189.83		10.6.20.231	Test IPS	8080	web-browsing	allow	threat	2.0k	pan-mgmt-sts-0
	start	12/22 12:42:17	trust	untrust	192.168.189.83		10.6.20.231	Test IPS	8080	web-browsing	allow	n/a	373	pan-mgmt-sts-0
	start	12/22 12:42:07	trust	untrust	192.168.189.83		10.6.20.231	Test IPS	8080	web-browsing	allow	n/a	373	pan-mgmt-sts-0

Figure 7.69: Logs about the detected and blocked threat

Finally, in order to test the URL filtering profile defined in our security policy, I simply had a pod connected to the firewall contact the URLs *www.gazzetta.it* and *www.ebay.it* set in the customised URL profile via curl: the result was that the connection to these external sites was blocked and the firewall responded with a warning HTML page (the 7.70 shows the response for www.gazzetta.it). In addition, I tried to contact some URLs stored in the malicious website's CN series database (in this case a couple of Palo Alto test URLs) and the connection was again blocked and the response returned was the same (Figure 7.71). In addition, the corresponding logs were generated, showing what happened (Figure 7.72).

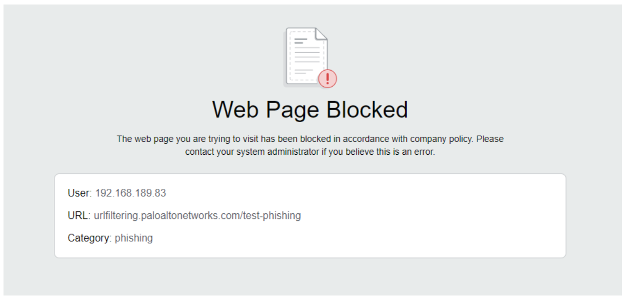


Web Page Blocked

The web page you are trying to visit has been blocked in accordance with company policy. Please contact your system administrator if you believe this is an error.

User: 192.168.189.83
URL: www.gazzetta.it/
Category: Ebay and Gazzetta - Test

Figure 7.70: Warning HTML page returned by the firewall - 2

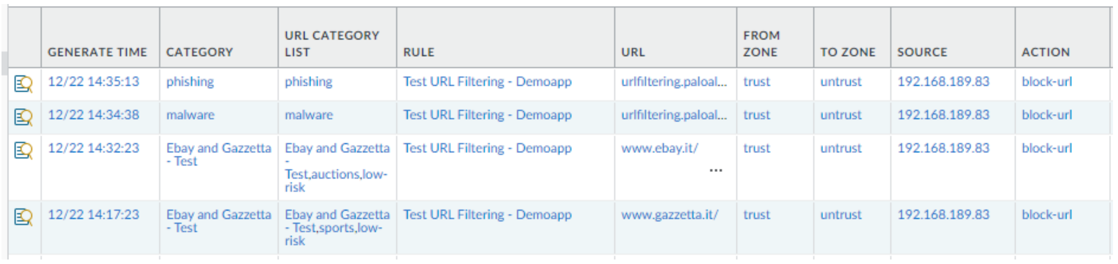


Web Page Blocked

The web page you are trying to visit has been blocked in accordance with company policy. Please contact your system administrator if you believe this is an error.

User: 192.168.189.83
URL: urlfiltering.paloaltonetworks.com/test-phishing
Category: phishing

Figure 7.71: Warning HTML page returned by the firewall - 3



	GENERATE TIME	CATEGORY	URL CATEGORY LIST	RULE	URL	FROM ZONE	TO ZONE	SOURCE	ACTION
	12/22 14:35:13	phishing	phishing	Test URL Filtering - Demoapp	urlfiltering.paloal...	trust	untrust	192.168.189.83	block-url
	12/22 14:34:38	malware	malware	Test URL Filtering - Demoapp	urlfiltering.paloal...	trust	untrust	192.168.189.83	block-url
	12/22 14:32:23	Ebay and Gazzetta - Test	Ebay and Gazzetta - Test,auctions,low-risk	Test URL Filtering - Demoapp	www.ebay.it/ ...	trust	untrust	192.168.189.83	block-url
	12/22 14:17:23	Ebay and Gazzetta - Test	Ebay and Gazzetta - Test,sports,low-risk	Test URL Filtering - Demoapp	www.gazzetta.it/	trust	untrust	192.168.189.83	block-url

Figure 7.72: Logs about the denied URLs tests

This demonstrated that the CN series capabilities allow for (at least partially) a zero-trust network with very strong network security PEPs that not only provide fine-grained authorisation policies, but also advanced capabilities that allow for the detection and blocking of certain network threats even if the connection was actually authorised at first. The lack of an integrated identity system for workloads within the cluster and traffic encryption, however, is the only negative point of this solution.

Chapter 8

Conclusions And Future Works

The starting objectives of the thesis were to attempt an implementation of Zero Trust Network models within Kubernetes clusters, and analyse and adopt the security functionalities provided by modern tools available on the market such as Service Mesh and Containerized Firewalls. After implementing and testing Istio and the Palo Alto CN-Series functionalities, some considerations were made, based on the results obtained.

Taking advantage of the sidecar pattern and Envoy proxies, Istio allows TCP communications to be protected by default via mTLS, regardless of when and where pods are executed, and without requiring any explicit configuration of the application: in this way, any plain TCP connection is automatically switched to a TLS channel, guaranteeing the confidentiality, authentication and integrity of every message exchanged, and thus achieving protection against eavesdropping, tampering and various types of man-in-the-middle attacks. Furthermore, thanks to the integrated CA and automated key and certificate management, Istio provides each pod with a reliable and verifiable digital identity, enabling the PEP to implement and execute a strong authentication mechanism. Unfortunately, these identities are based on the Service Account mounted by each pod, so care must be taken with any (possibly malicious) misconfigurations during the deployment phase, ensuring that a different Kubernetes Service Account is created and mounted for each service.

Thus, to ensure that only authorised entities can access a given resource within a pod, it is possible to successfully utilise the identity-based authorisation policies of Istio, extracting them from mTLS certificates and subsequently using these identities to match the defined authorisation policies. In addition, finer-grained policies can be implemented for incoming HTTP traffic, granting access to a specific HTTP endpoint only to certain workloads or end-users. In addition, network visibility is achieved for services within the network with some connection details, providing the ability to monitor operations within the cluster. However, some possible limitations of the Istio solution should be highlighted. First of all, the Envoy is only able to intercept TCP traffic, allowing non-TCP traffic to freely enter and leave the pods. This means that it is not possible to gain visibility and control over this type of traffic. The level of visibility provided by Istio on non-HTTP traffic is also limited, as any other TCP traffic is simply treated as plain TCP, and therefore no details are provided on the specific type of L7 traffic exchanged, and visibility is limited to L4 of the OSI layer. In addition, the proxy does not have full L7 visibility over HTTP traffic, but only over its headers (i.e. methods, paths, authorisation tokens): however, using the WebAssembly (Wasm) it is possible to extend the proxy's functionality and implement WAF capabilities, seeking full content visibility (but this is left to possible future work). In addition, server-first protocols are not supported by authorisation policies, since the latter operate server-side on the incoming traffic, while the former require that the first bytes after acceptance of the TCP connection are actually sent by the server (i.e. outgoing traffic). Finally, the most relevant consideration to take into account is that, since the application and sidecar containers run in the same network/process namespace, due to misconfiguration or poor implementation, the application may have the ability to remove the redirection rules and remove, alter, terminate or replace the sidecar proxy: this would allow a pod to intentionally bypass its sidecar for outgoing traffic or intentionally allow incoming traffic to bypass its sidecar.

On the other hand, the CN series does not encrypt traffic or provide a strong identity to workloads. By default, the containerised firewall simply relies on the IP addresses assigned to each K8s pod or service to enforce policies. However, these containerised firewalls can be configured to act as a TLS gateway for inbound or outbound traffic, using an appropriate certificate, and then to decrypt the traffic and express authorisation policies by also exploiting client/server identity. In addition to this, the CN series enables strong workload-to-workload micro-segmentation, based not only on source-destination IP pairs, but also on specific applications and service types. The level of observability achieved within any network traffic through these solutions is very high: full L7 inspection and visibility are achieved on any type of traffic, including non-TCP traffic. Moreover, the NGFW capabilities provided by the firewall allow north-south and east-west traffic to be protected, performing deep packet inspection and providing advanced protection against various threats, mitigating also their lateral spread. In addition, enhanced security is provided for each connected pod: since it is deployed as a completely separate node within the cluster and traffic redirection is implemented when the pod's network interface is created, there is no way around this PEP unless a product-specific vulnerability is discovered and exploited. However, both solutions provide a consistent and portable security posture and apply their security policies to any asset and workflow even when these are moved between corporate and non-company infrastructures, as the CN series can also be easily deployed on cloud-provider infrastructures, even overcoming the limitations of perimeter security solutions on such infrastructures. Both provide authorisation mechanisms to grant access with minimal privilege to every resource within the K8s cluster, realising micro-segmentation in different ways, and somehow limiting the lateral spread of threats.

Overall, in the end, I can say that both solutions alone do not entail a strong implementation of the zero-trust architecture, but they are a good starting point: each has some security features that are useful to achieve part of the complete zero-trust principles, and thus complement those of the other. This, however, was only the first step towards zero trust in these environments: future work will be done trying to integrate these two solutions together, exploiting the principle of defence in depth, and thus hopefully achieving a strong zero trust network in Kubernetes environments. Other future works will specifically address endpoint security within a Kubernetes cluster: the use of tools such as EDR (Endpoint Detection and Response) and XDR (eXtended Endpoint Detection and Response) are extremely useful nowadays within a network, as they offer real-time security monitoring and threat response capabilities, being able to detect patterns of behaviour that could indicate signs of malicious activity on a particular endpoint. As the integration of these security solutions within the network is important for threat detection, their deployment within Kubernetes modern environments (i.e., containers, pods and worker nodes) also becomes crucial to achieving complete Zero Trust and visibility.

Bibliography

[1] Google. *Google is a Leader in the 2023 Gartner® Magic Quadrant™ for Container Management.* 2023. URL: https://cloud.google.com/blog/products/containers-kubernetes/a-leader-in-2023-gartner-magic-quadrant-for-container-management.

[2] CNCF. *CNCF 2022 Annual Survey.* 2022. URL: https://www.cncf.io/reports/cncf-annual-survey-2022/.

[3] IBM. *What are microservices?* -. URL: https://www.ibm.com/topics/microservices.

[4] Altassian. *Microservices vs. monolithic architecture.* -. URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

[5] Gartner. *Microservices Architecture: Have Engineering Organizations Found Success?* 2023. URL: https://www.gartner.com/peer-community/oneminuteinsights/microservices-architecture-have-engineering-organizations-found-success-u6b.

[6] Cloud Native Computing Foundation. *Concepts Overview.* 2023. URL: https://kubernetes.io/docs/concepts/overview/.

[7] VMware. *Why use containers vs. VMs?* 2023. URL: https://www.vmware.com/topics/glossary/content/vms-vs-containers.html.

[8] Dinesh Kumar Ramasamy. *Life of a Packet in Kubernetes.* 2020. URL: https://dramasamy.medium.com/life-of-a-packet-in-kubernetes-part-1-f9bc0909e051.

[9] Google. *What is Kubernetes?* 2023. URL: https://cloud.google.com/learn/what-is-kubernetes.

[10] CNCF. *Kubernetes Components.* 2023. URL: https://kubernetes.io/docs/concepts/overview/components/.

[11] IBM. *What is Kubernetes?* 2023. URL: https://www.ibm.com/topics/kubernetes.

[12] CNCF. *Pods.* 2023. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[13] CNCF. *Sidecar Containers.* 2023. URL: https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/.

[14] CNCF. *Deployments.* 2023. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[15] CNCF. *ReplicaSet.* 2023. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

[16] CNCF. *Namespaces.* 2023. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[17] CNCF. *Service Accounts.* 2023. URL: https://kubernetes.io/docs/concepts/security/service-accounts/.

[18] Tigera. *Kubernetes Networking: The Complete Guide.* 2024. URL: https://www.tigera.io/learn/guides/kubernetes-networking/.

[19] Cloud Native Computing Foundation. *Services, Load Balancing, and Networking.* 2023. URL: https://kubernetes.io/docs/concepts/services-networking/.

[20] Tigera. *Kubernetes CNI Explained.* 2024. URL: https://www.tigera.io/learn/guides/kubernetes-networking/kubernetes-cni/.

[21] Cloud Native Computing Foundation. *Service.* 2023. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[22] Cloud Native Computing Foundation. *Ingress.* 2023. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/.

[23] Cloud Native Computing Foundation. *Ingress Controllers.* 2023. URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.

[24] IBM. *Calico.* 2022. URL: https://www.ibm.com/docs/pl/cloud-private/3.1.2?topic=ins-calico.

[25] Sumudu Liyanage. *Kubernetes Networking With Calico.* 2022. URL: https://medium.com/@sumuduliyan/kubernetes-networking-with-calico-623f4583ae8d.

[26] National Institute of Standards and Technology. *Zero Trust Architecture.* 2020. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf.

[27] Red Hat. *What's a service mesh?* 2018. URL: https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh.

[28] Dynatrace Corey Hamilton. *What is a service mesh? Service mesh benefits and how to overcome their challenges.* 2024. URL: https://www.dynatrace.com/news/blog/what-is-a-service-mesh/.

[29] The Istio Authors. *Architecture.* 2024. URL: https://istio.io/latest/docs/ops/deployment/architecture/.

[30] Jimmy Song. *Sidecar injection, transparent traffic hijacking, and routing process in Istio explained in detail.* 2022. URL: https://jimmysongio.medium.com/sidecar-injection-transparent-traffic-hijacking-and-routing-process-in-istio-explained-in-detail-d53e244e0348.

[31] The Istio Authors. *Security.* 2024. URL: https://istio.io/latest/docs/concepts/security/.

[32] Palo Alto Networks. *CN-Series Firewall for Kubernetes.* 2024. URL: https://docs.paloaltonetworks.com/cn-series/getting-started/cn-series-firewall-for-kubernetes.

[33] Palo Alto Networks. *Secure Kubernetes Workloads with CN-Series Firewall.* 2024. URL: https://docs.paloaltonetworks.com/cn-series/getting-started/cn-series-firewall-for-kubernetes/secure-kubernetes-workloads-with-cn-series.

[34] Palo Alto Networks. *CN-Series Core Building Blocks.* 2024. URL: https://docs.paloaltonetworks.com/cn-series/getting-started/cn-series-firewall-for-kubernetes/cn-series-core-building-blocks#id823f264f-a373-41c4-a1d0-54a3cd00953a.

[35] Palo Alto Networks. *Virtual Wire Interfaces.* 2024. URL: https://docs.paloaltonetworks.com/pan-os/10-1/pan-os-networking-admin/configure-interfaces/virtual-wire-interfaces.

[36] Cloud Native Computing Foundation. *Install Tools.* 2023. URL: https://kubernetes.io/docs/tasks/tools/.

[37] Cloud Native Computing Foundation. *Kubeadm*. 2023. URL: https://kubernetes.io/docs/reference/setup-tools/kubeadm/.

[38] IBM. *Calico*. 2023. URL: https://www.ibm.com/docs/pl/cloud-private/3.2.x?topic=ins-calico.

[39] Cloud Native Computing Foundation. *Images*. 2024. URL: https://kubernetes.io/docs/concepts/containers/images/.

[40] Cloud Native Computing Foundation. *Namespaces*. 2023. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[41] Cloud Native Computing Foundation. *Connecting Applications with Services*. 2023. URL: https://kubernetes.io/docs/tutorials/services/connect-applications-service/.

Acronyms

API	Application Program Interface
CA	Certification Authority
CD	Continuous Delivery
CI	Continuous Integration
CNI	Container Network Interface
CSR	Certificate Signing Request
DNS	Domain Name System
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol over Secure Socket Layer
IaaS	Infrastructure as a Service
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System

IPS	Intrusion Prevention System
K8s	Kubernetes
NGFW	Next Generation Firewall
OS	Operating System
PaaS	Platform as a Service
PDP	Policy Decision Point
PEP	Policy Enforcment Point
PKI	Public Key Infrastructure
RBAC	Role Based Access Control
SaaS	Software as a Service
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URL	Uniform Resource Locator
ZT	Zero Trust

List of Figures