uc3m | Universidad **Carlos III** de Madrid

Master Degree in Cybersecurity

Academic Year (e.g. 2024-2025)

Master Thesis

“Design and development of a zero-trust REST API in a cloud native environment”

Student, Jaime Romero Marín

Advisor, Alfonso De Jesus Perez Martinez

Madrid, September 2025

Abstract *- If the Internet is the information highway of industrial civilization, REST APIs have become the gateways through which this information finds its way to be consumed, stored, processed, and mined for value and capital gain. Every organization makes use of these interfaces in some way or another, from a simple personal website to the most ambitious AI enterprise. As a result, it makes them usually the first target of an attacker as well as the first place in which disruptions over a distributed system are detected. In this paper we will explore the state of the art of REST API security through the research and development of a secure architecture that follows zero trust principles as well as a proposed reference implementation. This reference implementation will simulate a critical military application operating in a cyberwarfare scenario in which it will need to maintain secure operation in a hostile network such as the open Internet using Zero Trust REST API design principles and protocols over a cloud native environment. A security test battery will also be provided to verify the correct implementation of this secure architecture.*

Keywords: Zero Trust, REST API, cloud-native, cloud, kubernetes, cybersecurity, architecture, network security, Istio, Golang, JWT, Rego, Open Policy Agent

I. Introduction

The objective of this assignment is to create a secure REST API using zero trust architectural patterns, principles and tenets with the purpose of exploring different technologies and security protocols that can be used in cloud environments, thus providing a reference implementation for further work in industry and academia.

In order to successfully develop a zero trust REST API we will follow a series of steps.

Our approach will begin by presenting on what is formally considered Zero Trust by NIST in order to create a set of formal security requirements that would qualify for a Zero Trust Architecture or ZTA. We will then present an example REST API application that will formalize a set of functional and security requirements that we will implement as a demo with specific open-source technologies that will allow us to explore the concepts of Zero Trust and secure architectures in a practical manner. This application will be designed to run in a cloud native environment using Kubernetes. We will then assemble a test battery to verify that all the security requirements have been satisfied and will finish with the conclusions of this practical project.

II. Previous Work

Other works of research have already been made since the first publication of NIST and the concept of Zero Trust has become the driving target for security engineering in the area. One of such publications is "Securing API-Based Integrations in Federated Cloud Architectures: A Zero Trust Perspective" [1]. In this article, one of the most interesting contributions is a framework for API Integrations that establishes a chain of processes that must be implemented on Zero Trust APIs. A diagram can be examined in figure 1.

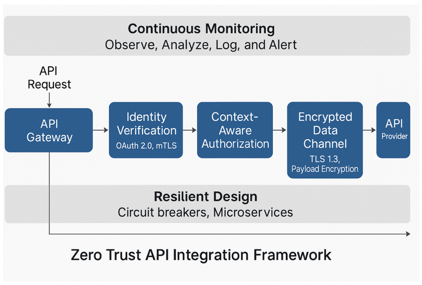

Continuous Monitoring
Observe, Analyze, Log, and Alert
API Request
API Gateway
Identity Verification
OAuth 2.0, mTLS
Context-Aware Authorization
Encrypted Data Channel
TLS 1.3, Payload Encryption
API Provider
Resilient Design
Circuit breakers, Microservices
Zero Trust API Integration Framework

Fig. 1. General Framework for Zero Trust API Integrations[1]

Similar concepts and definitions are also provided in "Designing Secure Microservices Architectures: Identity Management, API Security, and Zero Trust Approaches" [2], which introduces several technology recommendations such as Istio for the Service Mesh implementation, common features expected of API gateways as well as specialized back-ends for each front-end, and token management best practices. Further previous work can be found in "Cutting-Edge Practices for Securing APIs in FinTech: Implementing Adaptive Security Models and Zero Trust Architecture" [3] which examines the challenges of API security with a particular focus on the FinTech sector, proposing general principles such as "verify every request", "least-privilege access" and "microsegmentation" using the same security protocols such as JWTs, Mutual TLS and OpenID Connect, as exposed by other authors.

Definition of a REST API

REST APIs belong to a family of protocol definition architectures that are defined on RFC9205 to be under the following conditions [4].

- Uses the transport port 80 or 443, or
- Uses the URI scheme "http" or "https", or
- Uses an ALPN protocol ID that generically identifies HTTP (e.g., "http/1.1", "h2", "h3"), or
- Makes registrations in or overall modifications to the IANA registries defined for HTTP

There are many motivations for using these technologies for application development in the modern web such as the familiarity of stakeholders, availability of existing clients, server and proxy implementations, ease of use, availability of web browsers, presence of HTTP servers and clients in target deployments as well as its ability to traverse firewalls [4]. However, one of the most important reasons from a security perspective will be that we will be able to reuse existing security mechanisms such as authentication and encryption via TLS. From an application development perspective, the RFC9205 exposes the following advantages:

- *Generic semantics*: Potentially applicable for every resource and not specific of any particular application context with a well defined behavior for methods, header fields or status codes.
- *Links*: Which establishes a well defined namespace for accessing resources in a specific application by routing requests to different controllers on the application, as well as providing a natural mechanism for extensibility, versioning and capability management.

- *Rich Functionality*: Such as message framing, multiplexing, TLS integration, support for intermediaries, client authentication, content negotiation for format and language, caching, precise granularity of access control, partial content to selectively request part of a response, and the ability to interact with the application easily using a web browser.

All this advantages have popularized this way of building distributed systems on the web, raising concerns regarding their cybersecurity, making organizations such as OWASP to keep track of the most common attack patterns, listed on annex C.

Definition of a Zero Trust Architecture

According to NIST [5], *Zero Trust* is the term for an evolving set of cybersecurity paradigms that move defenses from static, network-based perimeters and policies towards focusing on users, assets and resources. Trust is defined as the set of assumptions by which a system makes decisions, in this case we are talking about authentication and authorization decisions. A *Zero Trust Architecture* or ZTA is one that assumes that there is no implicit trust granted to assets or user accounts based solely on their location or ownership. Thus, authentication and authorization functions are executed every time a session is established [5]. This presents itself in contrast to more traditional approaches such as perimeter security technologies such as firewalls and VPNs, which attempt to divide the world into trusted and untrusted domains. These systems are effective at protecting from attackers coming from the Internet, but drastically less effective at enforcing a high level of security against threats once they are able to breach into the trusted environment.

This gives rise to the concept of *implicit trust zone*, which represents the area or surface where all the entities are trusted at least to the level of the last policy enforcement point [5]. In the previous example, the implicit trust zone would be the internal network and the policy enforcement point would be the firewall or VPN server.

We will see in the following paragraphs that what zero trust architectures pretend to achieve is to reduce the size of the implicit trust zone as much as technically and practically possible. This is done by deploying strong authentication and authorization mechanisms on a policy enforcement point, which will be located as close to the resource to be protected as possible.

For example, instead of dividing the world into a trusted and untrusted network by the source of the packets, which can be easily spoofed in current IP networks, a next generation firewall could be integrated with a VPN to which individual hosts need to authenticate themselves to by using a per-client shared secret, establish a secure tunnel to the VPN server in which traffic is encrypted and routed to each host according to a network policy using the identities used in the authentication, which are much harder to spoof. In existing zero trust architectures, a variation that fits this description is the *resource portal model*. This setup drastically reduces the implicit trust zone from a whole network to a host-to-host level.

Formally speaking, Zero Trust security focuses on protecting discrete and individual information resources such as assets, services, workflows or network accounts, instead of whole networks. Each of these resources should have a specific and well defined identity that needs to be authenticated and authorized at every session or request [5].

For example, there are authorization protocols such as OAuth 2.0 PKCE that provide strong authorization and authentication guarantees that reduce the size of this implicit trust zone to individual processes in a host. This security model assumes that the host machine where this process is running might be infected by malware with read access via a side channel such as log files [6].

In other words, the main objective is to prevent unauthorized access to data and services coupled with making the enforcement of access control as granular as possible [5]. We have included a summary of the main Zero Trust tenets on annex B for reference.

In addition to these tenets, NIST defines a set of core zero trust logical components for any Zero Trust Architecture [5]. These components are basic terminology to be used, as they provide a common frame of reference for its design and development.

- *Policy Engine (PE)*: Responsible for processing the policy and the available data to make the decision to grant access to a resource for a given subject.

- *Policy Administrator (PA)*: Responsible for establishing and/or shutting down the communication path between a subject and a resource by generating session-specific authentication and authentication token or credential used by a client.

- *Policy Enforcement Point (PEP)*: Responsible for enabling, monitoring, and eventually terminating connections between a subject and an enterprise resource by receiving policy updates from the PA.

NIST also defines additional components to inform the PE with additional data sources from which we will focus on the most desirable ones for our application. These are a *network and system activity logs*, *enterprise public key infrastructure or PKI*, an *ID management system* and a *SIEM system*. We will mention these systems, but will leave their implementation out of the scope of this assignment as architecture decisions for the integration of this solution into a larger system.

With awareness of these concepts, NIST considers three architectural principles to design a ZTA [5].

- *Enhanced Identity Governance*: By using the authenticated identity of actors and their assigned attributes as the key component of policy creation.

- *Micro-Segmentation*: Placing individual or groups of resources on a unique network segment protected by a gateway security component as PA that controls a set of PEPs deployed as agents that shields the resources from unauthorized access or discovery.

- *Network Infrastructure and Software Defined Perimeters*: Involves using a software defined overlay network in which the PA acts as a network controller that sets up and reconfigured the network based on the decisions made by the PE, the clients request access via PEPs, which act as proxies.

Lets consider an example, we can define policies that apply to signed data regarding the users and resources of an application to sign assertions of properties about them. This establishes a trusted data structure to make policy decisions for either authentication or authorization.

This is the idea behind JWTs or JSON Web Tokens, which are basic JSON objects cryptographically signed when delivered to the client and verified upon arrival to a PEP. The fact that they are asigned allows for a verification that authenticates the client, the fact that the claims they contain are integrity protected allows the receiver of the JWT to make an authorization decision with the assurance that this request belongs to the user this JWT was previously delivered during user authentication. The purpose of enhanced identity governance is to minimize identity groups as much as possible, to the point of enforcing a policy per each identity.

In order to enable this, we will need to establish a certificate and identity management strategy in order for the entities of the system to sign the data they produce. When using certificates of identity, a PKI or Public Key Infrastructure is generally used. For the sake of simplicity in this project, we will just share the CA certificate between the clients and the server API, a more complex PKI is left as future work.

In the same application we can also use resource compartmentalization to achieve micro-segmentation, which is a logical conclusion from tenets ZTA:01 and ZTA:03 given that once we minimize the implicit trust zone of the resources and put a PEP in front of them, we obtain a unit of access control at which both authentication and authorization can occur. One of the main advantages of this concept is that it drastically reduces the capability of an attacker to enumerate targets as well as limiting its movement in the case exploitation is achieved, given that the adversary needs a successful authentication and authorization to access each new implicit trust zone breached. From a logging perspective, strong non-repudiation can be achieved once we can map the actions that an integrity-protected and authenticated identity such as a JWT has executed over the resources of a system, enabling rapid and precise investigations upon an incident. JWTs may be stolen if the communication is insecure, but they can be set with a validity period of minutes to reduce the attack window the adversary has after interception. If higher security is needed, JWTs can be configured with a counter, which further diminishes the capability of an adversary to replay it.

The third variation focuses on network security, the implicitly trusted zone becomes the network perimeter to be minimized and protected behind a PEP implemented as a network proxy. This implicitly trusted zone might be at the network level, at the host level, or at the process port level.

For the development of this thesis, we will use a mix of these variations, given that we want to design a secure network application with an access control model based on identity governance, running in a sandboxed cloud environment using a container orchestration platform. NIST defines several deployment models in which these logical variations can be implemented in different domains such as network environments or operating systems that perfectly represent the previous points which can be consulted in the annex B.

Definition of a Zero Trust REST API

NIST provides a set of guidelines for the protection of APIs in cloud native systems, recommending the deployment on an API Gateway that validates the NIST Zero Trust tenets [7]. NIST also provides a set of standardized security controls for a REST API, which are divided into basic and advanced, being the basic the mandatory standard for a secure REST API. The complete list for these security controls can be found at annex A.

To protect a REST API and apply the necessary set of policies that involve a ZTA architecture, NIST proposes the use of an API gateway [7]. The API gateway is the system responsible for mapping each request to its target endpoint, to which both authentication and authorization rules are applied. It is the front door to the back-end services of any application on the Internet.

For the development of this thesis, we will implement a distributed API gateway to create a secure REST API that verifies a ZTA using two key technologies, Kubernetes and Istio. The official diagram by NIST for this deployment model can be seen on figure 2.

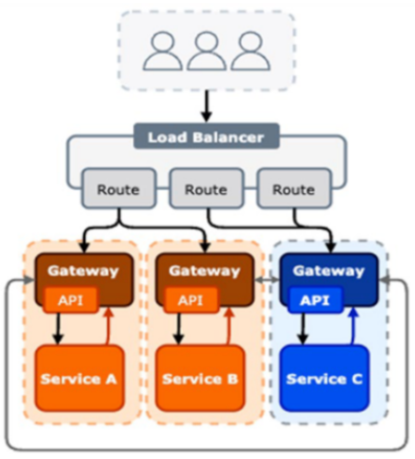


Fig. 2. Distributed Gateway according to NIST [7]

Identity-Based Segmentation

NIST proposes this concept as of the keys to successfully implement a REST API that verifies a ZTA. The idea is that every server should authenticate and authorize both the end-user identity and the client or software identity at every HTTP request, ideally at every single hop of the infrastructure. The following requirements must be met in order to successfully implement it.

- IBS:01 *Encryption In Transit*: To provide both data integrity and confidentiality and prevent unauthorized information disclosure or tampering.
- IBS:02 *Authentication of the calling software client*: To verify the identity of the software sending requests and avoid its spoofing by an attacker.
- IBS:03 *Authorization of the calling software client*: To verify that an authenticated client identity can do the action it pretends to execute.
- IBS:04 *Authentication of the end user*: Verify the identity of the entity behind the software to send the

request, which may a be human or a non-person entity.

- IBS:05 *Authorization of the end user*: To access the resources using the authenticated identity of the end-user to check that they are allowed to perform a certain operation on them.

For implementing IBS:01 we will use Istio to setup a service mesh for inter-cluster communications, as well as mutual TLS for the communication between the cluster services and the clients.

For IBS:02 and IBS:03 we will use x509 certificates for each kind of software client used, effectively implementing a kind of RBAC or Role-Based Access Control for the client identities, as user identities will be grouped by the client software they will use. These client identities will be used to segment the traffic to each of the different endpoints of the application in the cluster through the Subject Alternative Name or SAN and the Common Name or the CN of the x509 certificates using the Kubernetes Gateway API via HTTPRoutes and Istio as well as appending the XFCC Header to the requests once they are forwarded from the gateway into the service mesh.

For IBS:04 we will setup basic user and password authentication, based on a shared secret between the client and the server API. However, for more advanced deployments or federated applications OpenID Connect is recommended instead.

For IBS:05 we will use JWTs or JSON Web Tokens, which will be delivered to the client upon successful authentication and passed around in the requests as a Authentication Bearer HTTP header. This per-request authenticated identities will then be processed via an Open Policy Agent instance [8], which we will use as PE in the server API to enforce an authorization policy defined in Rego loaded into the service.

III. Case Study Application

For choosing the topic for the reference example of this zero-trust architecture, we have decided to go by the motto that it is better to be a warrior in a garden than a gardener in war. This is why we have chosen a military tech scenario as proof of concept of the application of a Zero-Trust application.

However, the knowledge extracted from this exercise is directly applicable to civil and peace-time use cases such as finance, the logistics of autonomous trucks, robotaxis, e-commerce deliveries or even a multiplayer online video game that provides strong anti-cheating guarantees for e-sport.

The example application will be a drone telemetry system for an hypothetical army in which the location of the drones at any given moment will be monitored.

We will consider three kinds of actors that will need to produce and consume information in the system with different degrees of confidentiality, the drones themselves, the pilots of the drone squads and officers.

The drones will just be able to consume orders in the form of a set of coordinates expressing the GPS location they need to be. They will also feed into the system the location they are currently in to create a feedback loop with the pilots. No more information about the battlefield should be available to a drone actor than its location and its target.

Drones are managed by a single Pilot, who is able to send orders to a squad of drones under its command (we are assuming these machines to have a high degree of autonomy). A Pilot actor will be able to read the locations of the drones it has been assigned to and will be able to point them to new targets. Any Pilot should only access a partition of the total information about the state of the battlefield related to his or her drones, without being able to see the information of any other Pilot.

An Officer actor will be able to access all the information related to the locations of the drones being managed by any Pilot, thus having access to a complete view of the battlefield. However, in this particular use case an officer will only be able to monitor the battlefield, without informing any of the Pilots or giving new targets to the Drones.

We will be using the Officer actor as the system administrator of the application, which will be used to provision and prepare the API with the user credentials for the Drones and the Pilots as well as the squad configuration between them.

In addition to this, we will define an adversarial actor, which will be external to these systems in order to test and verify the security of the architecture. The adversary will launch an attack battery against the system, first proving the access to the endpoints without TLS certificates captured, and then testing them in different situations for which the certificates for each actor have been disclosed, such as via drone capture or leaks from the Officer or one of the Pilots. Finally, we will consider the case in which a complete service and user identity has been spoofed in order to evaluate the authorization requirements.

We have provided an example in figure 3 to show the flows of drone location data in the system. The figure 4 represents the access control model for object properties for entities in the system. These diagrams will be used to create a fine-grained authorization policy using OPA and Rego.

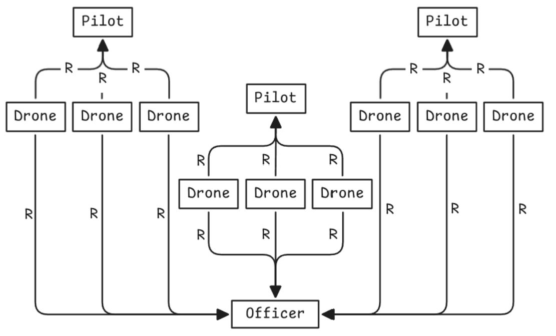


Fig. 3. Location data flows for a battlefield of 3 drone groups, 3 pilots and 1 officer

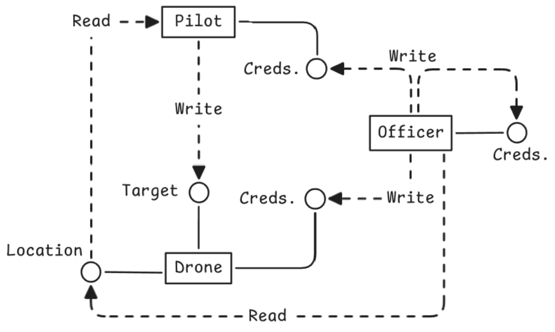


Fig. 4. Property access control model for entities in Drone API

IV. Security architecture

For the development of this application we will consider a DFD or Data Flow Diagram to express the flow of information through our system in order to identify threats to which mitigations need to be applied. This diagram can be found on the annex C on figure 11, as well as an enumeration of common threats provided by STRIDE, OWASP and NIST for API REST applications that can be used to deeply understand the threat model.

We will structure the exposition of the security architecture in terms of the measures taken to implement *identity-based segmentation* for our REST API. The same mitigation might overlap different security engineering functions, mutual TLS for example is useful for both implementing IBS:01 due to the fact that it provides a secure communication channel, but can also be leveraged for IBS:02 and IBS:03 due to the fact that it uses signed x509 certificates of identity for strong authentication, thus providing attributes such as the Common Name or the Subject Alternative Names that have high integrity guarantees, which allows us to use them on the authorization policy built using Rego and OPA.

In this reference implementation, we have provided mitigations for threat API:06 by achieving secure communications through the use of Mutual TLS and using a different certificate of identity per each of the clients as well as for the server. The communication inside the Kubernetes cluster is also protected via a service mesh managed using Istio, which automatically implements mutual TLS between the nodes of the mesh inside the cluster.

Once secure communication is achieved, we can consider the process of creating a strong authentication framework to mitigate threat API:02 via the use of Basic HTTP authentication and the mutual authentication that mutual TLS implicitly provides by establishing a common CA certificate for client and server certificate validation. In this case this is enabled by a shared CA certificate file that both the legitimate server and clients know.

Threats API:01 and API:05 are similar in the sense that they are authorization failures which are to be mitigated by the definition of a authorization decision policy and the availability of a PEP that can enforce it. In this application case study, the main resource to authorize access to is the location of the drones, while there are a couple of functions that only specific actors should be able to execute such as the provisioning of the credentials which should only be made by an officer. We will elaborate further in the following subsections.

NIST also proposes specific mitigations to validate a Zero Trust REST API for pre-runtime and runtime, which can be checked out at annex A.

Authentication mechanism

Previous authors make references to OpenID Connect as the main protocol for the IAM solution, however due to a restricted time budget and for the sake of focusing on the granular authorization system, the service mesh deployment details and the mutual TLS certificate deployment and API implementation, we decided to use a simpler authentication model based on a basic user and password exchange. However, OpenID Connect is a highly recommended solution for real world deployments and its incorporation in the reference implementation has been left for future work.

We will divide the authentication mechanism into two sections, client software authentication and end-user entity authentication. A diagram for the full authentication mechanism can be found in figure 5.

Client software authentication

We will be able to satisfy IBS:01 and IBS:02 by using mutual TLS to access the REST API.

This will force both the client and the server to authenticate each other using mutual TLS according to a common CA that needs to sign both the client and the server certificates used.

One of the decisions we had to make here is the size of the implicit trust zone for the identity segmentation that can be applied here.

On the sloppiest scenario, we can use a single certificate to authenticate any client that has it under its possession. This is the easiest option but also the one that provides the wider implicit trust zone, given that many different clients may use the same certificate as an identity umbrella, only requiring it to be captured once in order for an adversary to achieve client software authentication to the whole system.

The other extreme is that each single client instance has its own certificate, aligned with the unique identity of the user entity. This is the most desirable option from a zero trust perspective, given that it provides total end-user traceability at the expense of increasing the management complexity of both certificates and identities, thus requiring strong identity governance tools.

A more practical compromise, which we will follow on this thesis, is to have a client certificate for each kind of client, which aligns with the RBAC authorization policy we will want to implement for the client software authorization on the API.

- Drone client certificate
- Pilot client certificate
- Officer client certificate

Without presenting any of these certificates during mutual TLS, the client will simply be denied access by the API gateway.

End user entity authentication

For each entity on the system, both human and non-human, we will define username and password credentials in an automated manner to satisfy IBS:04.

After establishing a secure communication using mutual TLS, the entity shall present their credentials using HTTP Basic Authentication to a specific login endpoint, these credentials will be hashed with a salt in the server and compared against a previously provisioned set of hash credentials by an officer. No clear text credentials shall be stored in the process memory or database.

If there is a match, then the client software will receive a JWT for further per-request authentication, without needing the end user entity to present their credentials again until they need refresh.

JWTs follow a well known standard expressed in RFC 7519 which provides strong integrity guarantees via an authenticated digital signature using a secret only known by the API server. This allows the API to detect if a normal JSON object has been modified or tampered with, detecting when an attacker is attempting to replay it [9]. This allows the security architect of the API to trust the claims stored on the JWT to make both authentication and authorization decisions.

For the implementation of the API, we will use a middleware developed for the Echo web framework [10], which will automatically handle these tokens on each request.

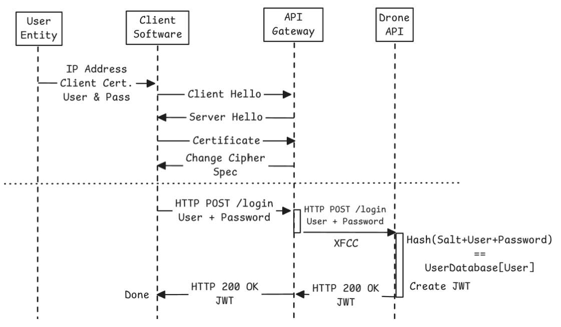


Fig. 5. Full authentication flow for both client software and end-user entity

Authorization mechanism

For the authorization strategy, we will consider two domains for authorization, one for the system-level for the deployment of the drone-api and another for application-specific authorization of each end-user.

Software Authorization

Regarding software authorization for IBS:03, we need to consider the security of the client software and the server.

Client Software Authorization

It is not possible to have complete control over what is being executed on the client without also having control of the hardware it is running on, which we will leave out of the scope of this thesis. The only thing we have to exercise some level of authorization is via the access provided by the client TLS certificates, which will allow us to authorize connections to specific endpoints.

The easiest way to implement this to implement an specific REST API for each role in alignment with the software authentication.

- Drone REST API specific actions, only authorized if the holder has a Drone cert.
- Pilot REST API specific actions, only authorized if the holder has a Pilot cert.
- Officer REST API specific actions, only authorized if the holder has an Officer cert.

A software identifying itself as a Drone client should never have access to the REST API endpoints of a Pilot or an Officer, thus implementing a mutually exclusive access control policy.

This can be done in a variety of ways, we decided to use the XFCC Header to pass the client certificate presented to the API gateway to the backend Drone API and parse the CN of the certificate to include it in our Rego authorization policy.

Server Software Authorization

Regarding software authorization of the drone-api, we will leverage the authorization framework delivered by Kubernetes, Linux containers and the Linux kernel in order to enforce the minimum privilege principle.

This involves using the smallest set of container Linux capabilities required for the process to execute its functions. The container image used for the drone API shall be the Scratch, which is an alias for the most basic and bare bones container image available in order to mitigate against living-of-the-land and supply-chain attacks. The API server written in Go will be a single statically linked binary that holds up everything needed for running the application. In addition to this, the files system of the running container will be strictly-read only, without the possibility of installing additional software once the server is deployed, and as a non-privileged user. These countermeasures are a second line of defense for a successful exploitation of the API. At the cluster level, this involves providing a K8S Service Account, defining a specific Role with minimal access to the K8S API and a corresponding Role Binding between the two for the pods of the drone-api deployment, which will limit the interaction with the Kubernetes API to strictly what is required. The deployment for the drone-api will also be restricted to its own Kubernetes namespace, limiting the access to cluster-wide resources.

End user authorization

In this section we will focus on the fine-grained authorization policy for the end user according to application requirements.

To achieve this, we will use OPA and define a Rego authorization policy. This allows the application developers to decouple authorization policy from system implementation, allowing for the establishment of an uniform access control policy. In most applications the authorization and access control policy is coupled to the application implementation right into the code. Over time the code base evolves and changes are made, which might lead to parts of the application not to follow the same authorization rules, leading to an heterogeneous authorization policy and security risks such as API:01, API:03 and API:05. Centralizing the authorization policies into a single PE assures that the same policy is applied to all the points of the application in which authorization is required. Decoupling authorization logic from the application logic also has the advantage of improving testing of the authorization policy itself, controlling for possible edge cases that may break authorization.

The authorization model used here will be ABAC, given that the OPA policies executed by the PE will take into account the identity attributes in the JWT to derive if the action is authorized or not. The policy is also taking into account the kind of operations being executed over the data, in order to mitigate against threat API:05 as well as API:06 and make sure not only the users have access to the object properties they should, but also to execute the actions they are authorized to.

Given that the main object level to which the policy is applied are drones, we will use this policy to return the identifiers of the drones the user can access. If the policy returns no drones, then it is assumed that the operation is not allowed.

For the actual implementation of the end-user authorization, we will use the Open Policy Engine, which is a general-purpose PE that is able to execute a set of security polices expressed in a *policy specification language* called Rego [8]. This Policy Engine will be executed as a library inside the drone-api. The Rego policy is informed by the authorization model and the input object, which contains the data that the policy will use to execute a policy decision.

The input object is essentially a JSON object that can implement any kind of schema that can then be used to develop an arbitrarily complex policy. It can be static or dynamically generated at runtime in order to implement context-dependent authorization policies, which is preferable from a Zero-Trust perspective due to the fact that these policies can be informed and customized according to real-time data and events.

Lets enumerate here the specific authorization requirements for the determination of which drone locations should a specific user access.

- A drone should only be able to publish its location and consume its target, never the other way around.
- A pilot should only be able to publish the target of the drones assigned to it and read the location of the drone, and never the other way around.
- An officer will only be able to access all the drone locations, as well as exclusive provisioning the API.

In addition to this, we are leveraging X-Forwarded-Client-Certificate or XFCC headers to inform the back-end about the client certificate presented at the API gateway and using it in the authorization decision of our logic. After the JWT authentication, we can fetch the role of the user and check if the certificate matches with it. If it doesn't, we assume this is a spoofing attempt using stolen credentials (API:02) or a security misconfiguration (API:08), thus denying access to the REST API.

These policies would be implemented by the Rego policy exposed in annex E, which is then run using Open Policy Agent as a library on the REST API or by delegating the authorization decision to a specialized service. This way, we can consider IBS:05 to be successfully implemented.

Rate limiting mechanism

The mitigations available for API:10 are dependent on where the consumption of the API is occurring. If within the cluster, Istio provides functionalities for rate limiting and circuit breaking mechanisms for the service mesh. If the consumer is outside the cluster, such as a abusive clients or an attacker, then the policy engine relies on the certificate and the user identities to exercise rate limiting or blocking. If a DDoS attack is targeting an unprotected endpoint such as the Login, then the only way to block an abuse such as an automated bot attack is to setup a Web Application Firewall in front of it in a real-life deployment, which is left as future work.

Request and Response Schema validation

For the request and response schema we will use OpenAPI, an *interface description language* which will solve multiple problems. First, it is a REST API specification language that can be used for both the design, documentation and versioning of the API, which is one of the key requirements for API security according to NIST [7] in order to mitigate the threat API:09 as well as for the documentation of the REST API. The endpoints, methods, request, responses data structures and other objects relate to the model of the application, such as the coordinates or the provisioning of the battlefield can be mapped.

The second utility of using OpenAPI is that it will allow us to generate stubs for the client and the server for a multitude of programming languages, increasing development speed of the system and its alignment and as well as automating the validation of each of the primitive types of the API. For this thesis we will use oapi-codegen, which allows us to generate the server and client stubs for the HTTP server using the Echo web framework and HTTP clients as well as request and response validation during API runtime, thus allowing us to satisfy the requirement REC-API-10 and REC-API-13 for API runtime protections.

We have provided an example of the OpenAPI definition of the provisioning endpoint used by the officers of the drone API in annex D.

System Architecture

Two key technologies used for system design are Kubernetes and Istio. While Kubernetes is the de facto standard for cloud engineering given that it has emerged as the best technology to manage workloads by leveraging container orchestration [11], Istio provides complementary features that fall outside the scope of Kubernetes specific to traffic control, reliability and security such as mutual TLS communication between services inside the cluster, API and ingress gateway configuration, XFCC header management, circuit breaking, load balancing, traffic shaping, service-to-service authentication and authorization, rate-limiting and request tracing over the service mesh [12].

It is important to point out from a threat model perspective that using an Istio service mesh is not free, as any attacker that can manipulate the istio-system namespace will be able to take full control of the mesh itself [13]. This why access in restricted in the Kubernetes cluster by segmenting the istiod deployment, the REST API itself and the API gateway each their own namespace. This is a logical separation that limits access to the cluster API resources. In addition to this, we have defined a REST API Role with minimal access control in the cluster and binded the service account of the REST API to it as a countermeasure to a successful exploitation attempt in the REST API.

Service Mesh

The motivation for implementing a service mesh in this project is that a multi-node Kubernetes cluster might have its internal traffic traverse through many kinds of networks, both public and private with different degrees of trust. The best way to secure this traffic is to encrypt it end to end via a service mesh, which effectively establishes the *Enclave Gateway* model described on annex B as one of the Zero Trust architectures.

Kubernetes has provided the ingress object since the beginning of the project, later versions define the Kubernetes API Gateway, which we will use to create the definition of the API Gateway given that future versions of Istio will use this resource definition to implement Ingress gateways [14].

The Kubernetes API Gateway divides the configuration of the load balancer into three new entities.

- *The GatewayClass object*: Managed by the cloud infrastructure provider.
- *The Gateway object*: Managed by the cluster or API gateway team.
- *The HTTPRoute object*: Managed by the microservice developers to define each of the REST API endpoints.

Istio implements the GatewayClass and sets up an Envoy proxy as the API Gateway, which is then injected in the cluster namespace where the Gateway object is declared. This deploys Envoy proxy under the hood to route the traffic according to the definitions of different HTTPRoute objects, to transport the requests arriving to the API Gateway of the specific HTTP endpoint. This communication occurs inside the Istio service mesh via a network of Envoy proxies, in our reference implementation traffic arrives to a sidecar in the Drone API pod where the traffic is authenticated and decrypted at the cluster level then forwarded to the Drone API. In addition to this, we have configured Istio to enable the XFCC or X-Forwarded-Client-Cert header, which allows us to send the certificate used by the client on the mutual TLS connection to the backend API, so that we can use it in the authorization policy.

Resource quota limits

In order to limit the resources consumed during the operation of the API, resource limits on the Kubernetes deployment have been established as well as a basic HPA, preparing for future work regarding the scalability of the service once a persistent data layer is provided.

Error reporting pattern

We will use standard HTTP codes to report errors in the system to the clients. However, we have changed 404 codes to 401 to avoid information disclosure on the API by an adversary with a stolen certificate and account. The reason for this is that the distinction between existence and unauthorized access can itself be a kind of information leak the adversary might leverage to obtain information on the system such as number of drones, to which pilots they belong as well as information about the officer or officers managing the system.

The main error reporting system for the administrators of the cluster will be logs written by the container processes involved, such as the API gateway, the envoy proxy and the drone api.

Logging and monitoring infrastructure

Logging is strongly dependent on the cloud administrators and the specific domain of the application. In this assignment we will consider logging within the framework of Kubernetes as every cloud has its own integrated solution, as well as being easily integrated with proprietary ones such as New Relic, Datadog, Dynatrace, AWS Cloud-

Watch and many others that are listed by Gartner [15] under the market for observability platforms. With this we want to make it clear that this is a big market and that a systematic review of all options is outside the bounds of this thesis.

The Istio command istioctl also provides a multitude of observability options for more complex service meshes and microservice architectures. It allows seamless integration with Envoy Proxy [16], Graphana [17], Jaeger [18], Kiali [19], Prometheus [20], Apache Skywalking [21] and Zipkin [22]. An exposition for the right configuration of these tools is left as future work.

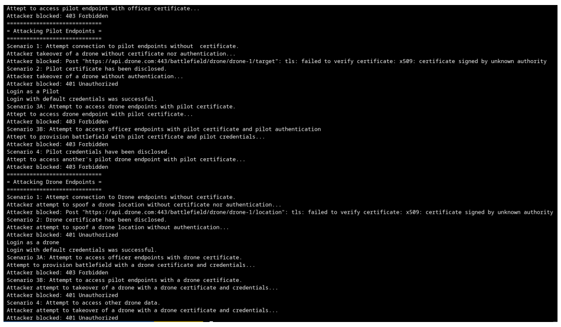
V. Evaluation And Testing

For evaluating the security architecture we have created a new kind of client that the adversary would use to breach this specific application that we will simply call the attacker client or attacker-cli.

It implements an attack battery in which different levels of breach and access for different roles is assumed. The idea is that the system needs to keep the enforcement of the authorization policy as individual credentials are captured.

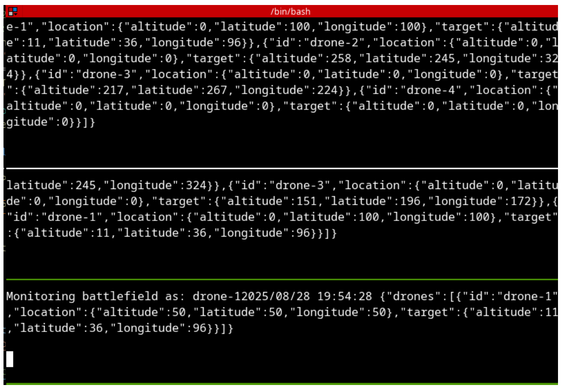
- *Scenario 1*: Attempt connection to the role endpoints without a certificate. The expected behavior is for the connection to be reset at the API gateway due to mutual TLS failure.
- *Scenario 2*: The software client certificate for a particular role has been disclosed by the adversary, but access to the drone API endpoints other than the login endpoint result in HTTP 401 Unauthorized.
- *Scenario 3*: Both the client certificate and the user credentials are disclosed to the adversary, but access to unauthorized operations for that role are forbidden and result in a HTTP 403.
- *Scenario 4*: Applied only to pilots and drones, both the client certificate and the user credentials are disclosed to the adversary, but access to other pilots for the case of pilots and other drones for the case of drones are forbidden and result in a HTTP 403 or 401.

On figure 6 we can observe the execution of this attack battery and the observation that all the authentication and authorization attacks for each scenario result in some kind of failure for the attacker. Which validates that our authentication and authorization systems are working at different levels of compromise.



```
Attept to access pilot endpoint with officer certificate...
Attacker blocked: 403 Forbidden
=============================
= Attacking Pilot Endpoints =
=============================
Scenario 1: Attempt connection to pilot endpoints without  certificate.
Attacker takeover of a drone without certificate nor authentication...
Attacker blocked: Post "https://api.drone.com:443/battlefield/drone/drone-1/target": tls: failed to verify certificate: x509: certificate signed by unknown authority
Scenario 2: Pilot certificate has been disclosed.
Attacker takeover of a drone without authentication...
Attacker blocked: 401 Unauthorized
Login as a Pilot
Login with default credentials was successful.
Scenario 3A: Attempt to access drone endpoints with pilot certificate.
Attept to access drone endpoint with pilot certificate...
Attacker blocked: 403 Forbidden
Scenario 3B: Attempt to access officer endpoints with pilot certificate and pilot authentication
Attept to provision battlefield with pilot certificate and pilot credentials...
Attacker blocked: 403 Forbidden
Scenario 4: Pilot credentials have been disclosed.
Attept to access another's pilot drone endpoint with pilot certificate...
Attacker blocked: 403 Forbidden
=============================
= Attacking Drone Endpoints =
=============================
Scenario 1: Attempt connection to Drone endpoints without certificate.
Attacker attempt to spoof a drone location without certificate nor authentication...
Attacker blocked: Post "https://api.drone.com:443/battlefield/drone/drone-1/location": tls: failed to verify certificate: x509: certificate signed by unknown authority
Scenario 2: Drone certificate has been disclosed.
Attacker attempt to spoof a drone location without authentication...
Attacker blocked: 401 Unauthorized
Login as a drone
Login with default credentials was successful.
Scenario 3A: Attempt to access officer endpoints with drone certificate.
Attempt to provision battlefield with a drone certificate and credentials...
Attacker blocked: 403 Forbidden
Scenario 3B: Attempt to access pilot endpoints with a drone certificate.
Attacker attempt to takeover of a drone with a drone certificate and credentials...
Attacker blocked: 401 Unauthorized
Scenario 4: Attempt to access other drone data.
Attacker attempt to takeover of a drone with a drone certificate and credentials...
Attacker blocked: 401 Unauthorized
```

Fig. 6. Output returned by the attacker-cli command



```
/bin/bash
e-1","location":{"altitude":0,"latitude":100,"longitude":100},"target":{"altitud
e":11,"latitude":36,"longitude":96}},{"id":"drone-2","location":{"altitude":0,"l
atitude":0,"longitude":0},"target":{"altitude":258,"latitude":245,"longitude":32
4}},{"id":"drone-3","location":{"altitude":0,"latitude":0,"longitude":0},"target
":{"altitude":217,"latitude":267,"longitude":224}},{"id":"drone-4","location":{"
altitude":0,"latitude":0,"longitude":0},"target":{"altitude":0,"latitude":0,"lon
gitude":0}}]}

latitude":245,"longitude":324}},{"id":"drone-3","location":{"altitude":0,"latitu
de":0,"longitude":0},"target":{"altitude":151,"latitude":196,"longitude":172}},{
"id":"drone-1","location":{"altitude":0,"latitude":100,"longitude":100},"target"
:{"altitude":11,"latitude":36,"longitude":96}}]}

Monitoring battlefield as: drone-12025/08/28 19:54:28 {"drones":[{"id":"drone-1"
,"location":{"altitude":50,"latitude":50,"longitude":50},"target":{"altitude":11
,"latitude":36,"longitude":96}}]}
```

Fig. 7. Output of officer, pilot and drone clients from top to bottom

VI. Future Work

Due to the time budget or complexity of implementation, several lines of work are still open and might be worth exploring in future R&D efforts.

References exposed in previous work presents OpenID Connect as the recommended federated authentication solution, future work on this thesis involves expanding the reference implementation with this protocol to authenticate the users. Regarding the hardening of the login endpoint, a solution against automated brute-force attacks is also left as future work.

The Istio service mesh is a fundamental part of the security in this setup and its integrity must be maintained. One possible line of improvement would be to deploy the Istio control plane on a different Kubernetes cluster from the one in which the managed workloads are being executed, drastically improving the security of the service mesh beyond a compromise of the Kuubernetes cluster where the services are running. This is a setup documented by Istio [23] but which would get outside the scope of the objectives of this assignment and which would probably only be worth the overhead in management and deployment complexity in a multi-cluster large scale deployment such as the ones maintained by Amazon Web Services and Azure.

The TLS and mutual TLS security model are only effective when both the client and the server are able to authenticate the certificate by a common certificate authority. In this assignment this was achieved by means of a shared CA certificate, but in a real-scale deployment a PKI should be defined and deployed in other to scale creation, distribution and repudiation of certificates. An advanced certificate management solution would also open the possibility of reducing the implicit trust zone further by delivering an unique client certificate to each user of the system, providing stronger authentication and non-repudiation guarantees.

In a real deployment, unprotected endpoints such as the login should be protected using a WAF such as OWASP's Modsecurity [24]. This has been left out of the assignment due to the fact that it falls outside the implementation of the API itself. However, such a system could generate valuable metrics that might be considered in more complex Rego authorization policies, as well as the introduction of a SIEM.

In addition to all of this, the data of the REST API is currently being stored in the process memory, so a more scalable and efficient data management solution that maintains current Zero-Trust properties such as the use of an encrypted key-value or relational database is needed.

VII. Conclusion

On this thesis we researched the existing documentation on Zero-Trust architectures, documented the necessary requirements and principles used to implement one, and applied them to a reference application in order to demonstrate how existing open source and cloud technologies can be leveraged to implement such architectures while also addressing technical security solutions to well-known threats towards REST APIs. This reference implementation was them formally tested in order to prove the strength of both the authentication and authorization policies.

As a result, a reference Zero-Trust API that meets all the basic pre-runtime and runtime protections exposed in annex A was developed, establishing a proof-of-concept for future R&D efforts in industry and academia alike.

All the assets for this work can be found at a public Github repository for replication purposes `https://github.com/jairomer/cybersecurity-master-thesis`.

ANNEX A: RECOMMENDED BASIC AND ADVANCED PROTECTIONS FOR REST APIS ACCORDING TO NIST

NIST organizes REST API protections in two sections.

1. *API Pre-Runtime Protection*: Controls applied during design, development and testing.
2. *API Runtime Protection*: Controls applied per each request and response during the API operation.

Each section has two levels of protections.

1. *Basic Protections*: Should be mandatory and pursued immediately.
2. *Advanced Protections*: Require deeper traffic inspection and the security they provide usually involve certain tradeoffs such as radical increase in computation or complex management.

API Pre-Runtime Protections

Basic Protections

- REC-API-1 *All APIs must have an API specification*
- REC-API-2 *API specifications should use an Interface Description Language.*
- REC-API-3 *APIs should have a well defined request and response schema.*
- REC-API-4 *A centralized API governance framework shall be established.*

Advanced Protections

- REC-API-5 *APIs should have a request and response schema-based validations.*
- REC-API-6 *APIs should have required-permission annotated in the schema.*
- REC-API-7 *APIs should have their fields annotated with a semantic type.*
- REC-API-8 *APIs should have their runtime metadata in the API inventory.*

API Runtime Protections

Basic Protections

- REC-API-9 *API Communication Must Be Encrypted.*
- REC-API-10 *A general schema validation policy shall be applied to each request.*
- REC-API-11 *APIs shall have a robust authentication mechanism and correct processing of credentials.*
- REC-API-12 *The end-user and service identities shall be verified at each request.*
- REC-API-13 *Requests and response shall be validated before being processed by the business logic.*
- REC-API-14 *Incoming requests shall be authenticated, authorized, validated and finally sanitized in this particular order.*
- REC-API-15 *The API shall enforce resource usage limits.*
- REC-API-16 *The API shall enforce rate limiting to internal callers.*
- REC-API-17 *The API shall enforce fine-grained request and user blocking, without affecting a legal and non-abusive users.*
- REC-API-18 *The API access control processes shall be monitored.*

Advanced Protections

- REC-API-19 *Field-Level Validation Using API Schema Annotations.*
- REC-API-20 *API Schema Annotations For Authorization.*
- REC-API-21 *Semantic Field Logs and Monitoring.*
- REC-API-22 *Non-Signature Payload Scanning.*
- REC-API-23 *Design Against Resource Enumeration.*
- REC-API-24 *Rate-Limit Resource Enumeration Attacks.*
- REC-API-25 *Limit Sensitive Data Exposure In Response And Logs.*
- REC-API-26 *Block High Impact Requests.*

ANNEX B: REQUIREMENTS FOR ZERO-TRUST ARCHITECTURES

Tenets of Zero Trust

To create Zero Trust systems, the recommendation of NIST is follow a set of basic technology-agnostic tenets or requirements [5], which we will summarize here given that they are relevant for the execution of this thesis.

- ZTA:01 *All data sources and computing services are considered resources*: A network may be composed of multiple classes of devices.
- ZTA:02 *All communication is secured regardless of network location*: Access requests from assets located on enterprise-owned network infrastructure must meet the same security requirements as access requests and communication from any other non-enterprise-owned network.
- ZTA:03 *Access to individual enterprise resources is granted on a per-session basis*: Trust in the requested is evaluated before the access is granted, with the less privileges needed to complete the task.
- ZTA:04 *Access to resources is determined by dynamic policy, including the observable state of client identity, application/service and the requesting asset, and may include other behavioral and environmental attributes*: Examples of these attributes can be device characteristics, previously observed behavior, time and date, environmental attributes or different kinds of analytics in order to create policy rules, which is the set of access rules based on attributes that an organization assigns to a subject, data asset or application.
- ZTA:05 *The enterprise monitors and measures the integrity and security posture of all owned and associated assets*: No asset is inherently trusted and continuous diagnostics and mitigation.
- ZTA:06 *All resource authentication and authorization are dynamic and strictly enforced before access is allowed*: An enterprise should have Identity, Credential and Access Management and asset management systems in place, including multi-factor authentication for some or all resources.
- ZTA:07 *The enterprise collects as much information as possible about the current state of assets, network infrastructure and communications and uses it to improve its security posture*: An enterprise should collect data about each asset security posture, network traffic and access requests, process that data and use any insight gained to improve policy creation and enforcement.

Assumptions of a ZTA

NIST also defines a set of system architecture requirements to verify the implementation of ZTA that we will rephrase in a requirement specific language [5].

- NET:01 *Enterprise assets shall be assumed to have a basic network connectivity with basic routing and infrastructure.*
- NET:02 *Enterprise shall be able to distinguish assets in the network via issued credentials and information with strong data integrity guarantees.*
- NET:03 *The enterprise shall observe all network traffic as well as managing the metadata of the connection.*
- NET:04 *The enterprise assets shall never be reachable except via an authenticated connection through a PEP.*
- NET:05 *The data plane and the control plane of the ZTA's network shall be logically separated.*
- NET:06 *Enterprise assets shall only be reachable from a PEP component.*
- NET:07 *The PEP shall be the only component that accesses the PA as part of the business flow being secured.*
- NET:08 *Remote enterprise assets shall never need to access remote enterprise resources by traversing the enterprise network infrastructure first.*
- NET:09 *The infrastructure to implement the ZTA access decision process should be scalable to account for changes in load.*
- NET:10 *Some enterprise shall not reach certain PEP if the security policy states so.*

ZTA Deployment Models

It is worth noting that NIST defines several deployment models in which these logical variations can be implemented in different domains such as network environments or operating systems. For the purposes of this thesis, we will briefly mention the most interesting ones.

The first is the *enclave gateway model*, which is generally used in cloud native microservice architectures [5]. This model fronts the gateway to a resource enclave, to which access is granted through an agent with all the configuration needed to access the system. In this model, the agent may be an application installed on a desktop or mobile device that connects to a backend via an API gateway.

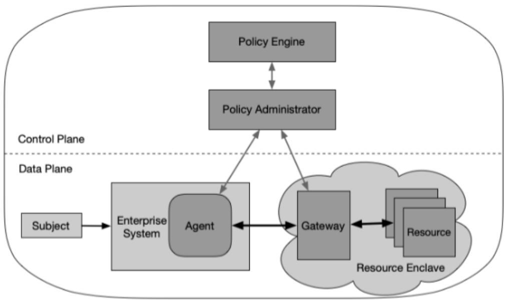


Fig. 8. Enclave gateway model according to NIST [7]

Another deployment model that NIST proposes is the *resource portal model*, which is the most similar to the majority of REST APIs on the web due to the fact that it is agentless, allowing for increased connectivity with clients by means of a web browser without the need of installing a heavy client. Instead, the policy enforcement point is applied at the gateway as requests arrive into a system. The main disadvantage of this model is that the owners of the system lose visibility and control over the users of the API while also exposing the API to the threats of the open internet[5].

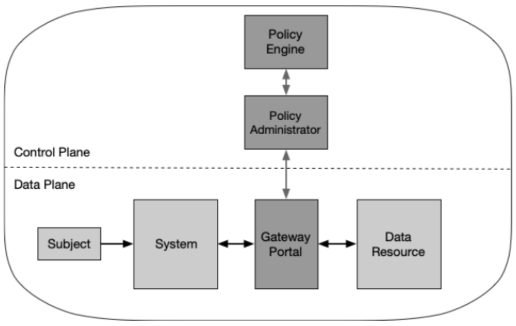


Fig. 9. Resource portal model according to NIST [7]

Finally, NIST also considers the *device application sandboxing model*, which leverages the compartmentalization principle to create secure architectures through resource segmentation. In this model the compartments may be virtual machines or container instances, and the main advantage is that if correctly implemented it will protect the system from compromised individual assets [5]. We will use this model on the topic of this thesis at the system level architecture, for which we will use both virtual machines for nodes and abstractions provided by Kubernetes such as namespaces and pods to orchestrate containers.

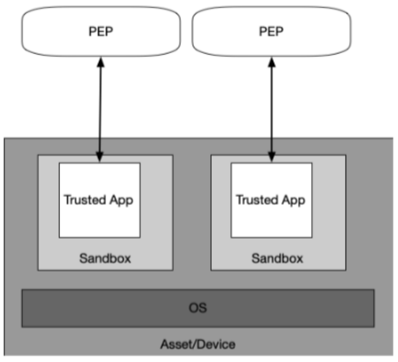


Fig. 10. Application sandbox model according to NIST [7]

ANNEX C: REST API CYBERSECURITY RISKS AND THREATS

In order to architect and develop a secure solution of any kind, we need to develop a threat model.

We will assume for the remaining of this thesis that an attacker can be either a human with malicious intentions or software with any degree of sophistication, from simple scripts to advanced AI, that behaves as what we could consider as malware or abusive behavior, and so we will not make any distinction. We will make the same assumption for the defender.

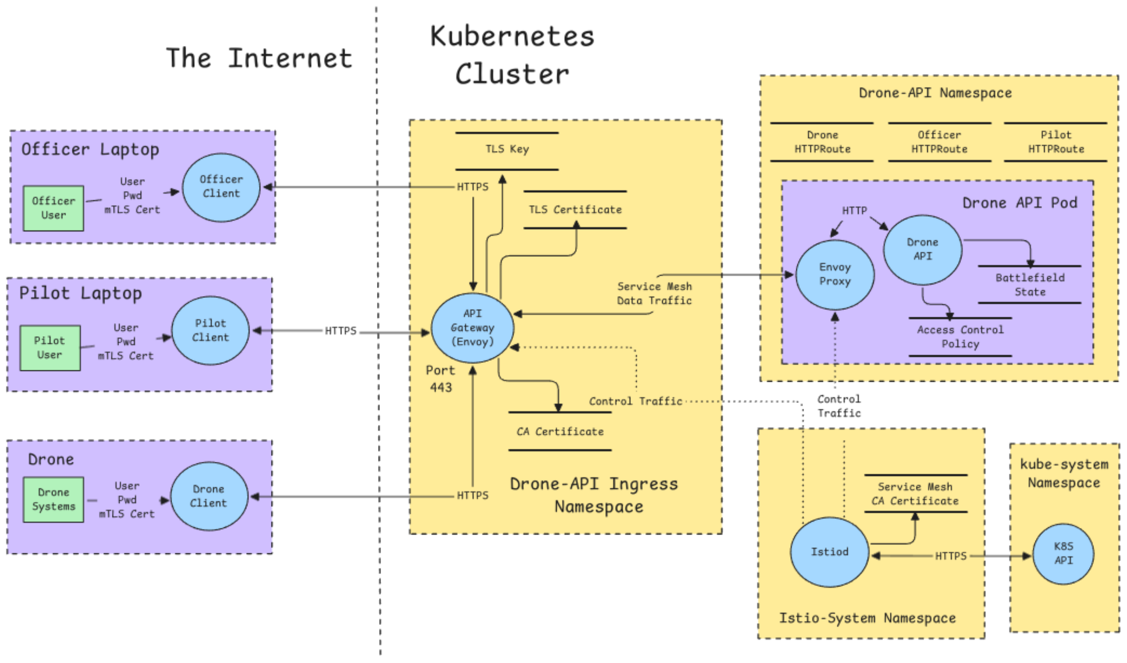


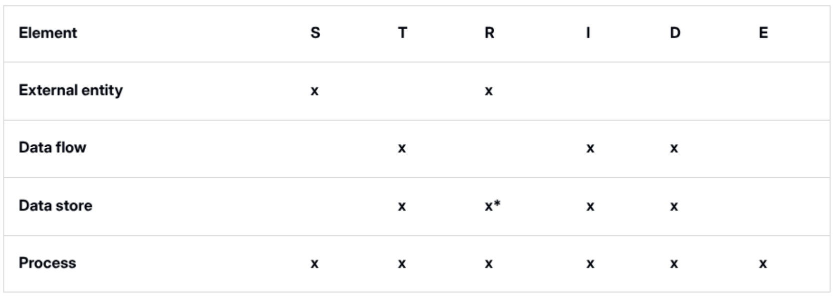
Fig. 11. Data Flow Diagram For The API Application

For categorizing the threats, we will use the general framework given by STRIDE [25], which involves:

1. *Spoofing:* An attacker manages to take an identity that does not correspond to them.
2. *Tampering:* An attacker manages to write or modify data that they should not be able to.
3. *Repudiation:* An attacker manages to negate the link between the identity they are using and a change in the state of a mutable system.
4. *Information Disclosure:* An attacker successfully manages to read data they should not be able to.

5. *Denial of service:* An attacker manages to deny access from a set of users to a given service or asset with varying degrees, from degradation to complete permanent unavailability.

6. *Elevation of privilege:* An attacker manages to achieve a higher level of authorization over the actions of the assets in a system that they should.



Element	S	T	R	I	D	E
External entity	x		x			
Data flow		x		x	x	
Data store		x	x*	x	x	
Process	x	x	x	x	x	x

Fig. 12. Mapping of STRIDE threats to DFD entities

As expressed in figure 12, each of the entities on the DFD diagram of the application have immediate security threats attached to them by default. In the following paragraphs we will also provide REST API specific threats to be considered.

Given that a significant number of websites on the Internet use REST APIs for interacting with their backend, much of the threats that affect them will also affect REST APIs. For this reason, OWASP monitors the state of API REST security as one of their concerns and provides a top 10 of security risks in 2023 [26] we will use in the threat model of our example application as well as the secure deployment of said application using cloud technologies.

- API:01 *Broken Object Level Authorization*: This is the risk of not correctly managing access control for the objects that need to be managed in the API, this leads to the realization of may different kinds of threats depending on the kind of excessive access that is given.

- API:02 *Broken Authentication*: This is the risk that a user is able to break authentication in some way that compromises the system ability to prove his identity from that of another user.

- API:03 *Broken Object Property Level Authorization*: On an application we might find different hierarchies, groups and domains for access control, as well as a distinction between administrative and regular functions. This risk represents the failure to correctly manage this complexity from a development perspective such as excessive permissions for workloads or accounts.

- API:04 *Unrestricted Resource Consumption*: Satisfying API request requires limited resources such as bandwidth, CPU, memory and storage, which when exhausted can lead to denial of service. Other resources to other APIs that are subject to costs per request may also imply an economic impact to the organization, impacting in operational costs.

- API:05 *Broken Function Level Authorization*: Similar to API4:2023 but applied to actions or functionalities on an applications instead of access to objects and their properties.

- API:06 *Unrestricted Access to Sensitive Business Flows*: Business flows are the functionalities that allow users to achieve a certain objective in the application, such as buying a ticket for a concert, which can be abused by bots that buy all the tickets to a concert in order to sell them into secondary markets.

- API:07 *Server Side Request Forgery/SSRF*: Which occur when an API is fetching a remote resource without validating the user-supplied URI. This allows an attacker to coerce the application to send a crafted request to a trusted location.

- API:08 *Security Misconfiguration*: APIs and the systems supporting them typically contain complex configurations to make the API more customizable. Even if the API itself is secure, it can be configured in an unsecured manner that enables a vector for exploitation by an attacker.

- API:09 *Improper Inventory Management*: APIs tend to expose more endpoints than traditional web applications, making proper and updated documentation very important. This risk also opens the door for an attacker to exploit what are known as shadow APIs, which are undocumented, or zombie APIs which are assumed to be deprecated or retired but still available on the application. Both of these classes can be problematic because they are outside the awareness of the API administrators and can have latent vulnerabilities subject to exploitation.

- API:10 *Unsafe Consumption of APIs*: Developers tend to trust data received from third-party APIs more than user input, adopting a weaker security standard. An attacker can compromise a third-party service instead of trying to compromise the target API directly.

NIST also provides a set of general threats towards REST APIs that we will use to define the ZTA, which overlap with most of the ones provided by OWASP [7].

One of the main risks that NIST brings up but OWASP ignores is:

- API:11 *Lack of Canonicalization of credentials and identities*: One of the main problems that emerge when trying to implement a ZTA comes from the fact that there exist a multiplicity of authentication and authorization standards such as certificates, SPIFFE identities, mutual TLS or JWTs. The solution is the standardization of the credentials that the application implemented at the edge of the system, in the API Gateway, before deploying additional controls.

ANNEX D: OPENAPI SPECIFICATION

We are providing here an example of an OpenAPI specification used in the reference application for illustration purposes for the reader.

The following is an example for the battlefield provision endpoint of the REST API.

```
/battlefield/provision:
    post:
      summary: An officer will use this endpoint to provision resources
               for a battlefield.
      operationId: BattlefieldProvision
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/BattlefieldProvision'
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/BattlefieldData'
        '403':
          description: Forbidden Access
        '400':
          description: Bad Request
```

This is an example of a data structure definition for the API.

```
    BattlefieldProvision:
      type: object
      properties:
        credentials:
          type: array
          items:
            $ref: '#/components/schemas/UserProvision'
        pilots:
          type: array
          items:
            $ref: '#/components/schemas/PilotProvisioning'
      required:
        - credentials
        - pilots
    UserProvision:
      type: object
      properties:
        user:
          type: string
        password:
          type: string
        role:
          type: string
          enum: ["officer", "pilot", "drone"]
      required:
        - user
        - password
        - role
    PilotProvisioning:
      type: object
      properties:
        id:
          type: string
          example: pilot-x
        drones:
          type: array
          items:
            $ref: '#/components/schemas/DroneData'
      required:
        - id
        - drones
```

ANNEX E: REGO AUTHORIZATION POLICY

This annex contains the authorization decision policy used in the reference API implementation.

```
package battlefield.authz

# default deny unless explicitly allowed
default allow = false

# A drone can only have access to its own data.
drones[drone] if {
    user := input.request.user.id
    role := input.request.user.role
    op := input.request.user.operation

    p := input.battlefield.pilots[_]

    role == "drone"
    p.drones[_] == user
    # Allowed operations
    op == { "get-target", "set-location", "get-battlefield"}[_]
    drone := user
}

# A pilot can access a drone if the drone belongs to them.
drones[drone] if {
    user := input.request.user.id
    role := input.request.user.role
    op := input.request.user.operation

    role == "pilot"
    p := input.battlefield.pilots[_]
    p.id == user
    # Allowed operations
    op == {"set-target", "get-battlefield"}[_]
    drone := p.drones[_]
}

# An officer can access data from all drones in the battlefield.
drones[drone] if {
    user := input.request.user.id
    role := input.request.user.role
    op := input.request.user.operation

    role == "officer"
    p = input.battlefield.pilots[_]
    # Allowed operations
    op == {"get-battlefield", "provisioning"}[_]
    drone := p.drones[_]
}

allow if {
    count(drones) > 0
}
```

BIBLIOGRAPHY

[1] A. Ramaswamy, “Securing api-based integrations in federated cloud architectures: A zero trust perspective,” *European Journal of Information Technologies and Computer Science*, vol. 5, no. 4, pp. 1–4, Jul. 2025. DOI: 10.24018/compute.2025.5.4.154. [Online]. Available: https://ej-compute.org/index.php/compute/article/view/154.

[2] A. Oluwaferanmi, “Designing secure microservices architectures: Identity management, api security, and zero trust approaches,” 2025.

[3] A. K. Bayya, “Cutting-edge practices for securing apis in fintech: Implementing adaptive security models and zero trust architecture,” *International journal of applied engineering and technology (London)*, vol. 4, pp. 279–298, Sep. 2022.

[4] M. Nottingham, *Building Protocols with HTTP*, RFC 9205, Jun. 2022. DOI: 10.17487/RFC9205. [Online]. Available: https://www.rfc-editor.org/info/rfc9205.

[5] Scott Rose, Oliver Borchert, Stu Mitchell, Sean Connelly, “Zero trust architecture,” National Institute of Standards and Technology, NIST Special Publication, 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf.

[6] N. Sakimura, J. Bradley, and N. Agarwal, *Proof Key for Code Exchange by OAuth Public Clients*, RFC 7636, Sep. 2015. DOI: 10.17487/RFC7636. [Online]. Available: https://www.rfc-editor.org/info/rfc7636.

[7] Chandramouli, Ramaswamy, “Guidelines for api security for cloud-native systems,” National Institute of Standards and Technology, NIST Special Publication, 2023. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-228.ipd.pdf.

[8] O. P. A. contributors, *Open policy agent (opa): A general-purpose policy engine*, https://www.openpolicyagent.org/, Open source, general-purpose policy engine unifying policy enforcement across the stack (CNCF graduated)., 2025.

[9] M. B. Jones, J. Bradley, and N. Sakimura, *JSON Web Token (JWT)*, RFC 7519, May 2015. DOI: 10.17487/RFC7519. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7519.txt.

[10] LabStack, *Echo-jwt (go package)*, version v0.0.0-20221127215225-c84d41a71003, Accessed: 2025-08-28, 2022. [Online]. Available: https://pkg.go.dev/github.com/labstack/echo-jwt@v0.0.0-20221127215225-c84d41a71003.

[11] K. Authors, *Kubernetes: Orquestación de contenedores para producción*, Accedido: 2025-08-28, 2025. [Online]. Available: https://kubernetes.io/es/.

[12] I. Authors, *Istio: Plataforma de malla de servicios de código abierto*, Accedido: 2025-08-28, 2025. [Online]. Available: https://istio.io/.

[13] The Istio Authors, *Istio security model*, https://istio.io/latest/docs/ops/deployment/security-model/, Accessed: 2025-08-13, 2025.

[14] F. Budinsky, *Getting started with the kubernetes gateway api*, Istio Blog, Using the Gateway API to configure ingress traffic for your Kubernetes cluster, Dec. 2022. [Online]. Available: https://istio.io/latest/blog/2022/getting-started-gtwapi/.

[15] Gartner Peer Insights, *Observability platforms reviews and ratings*, Gartner Peer Insights Webpage, Accessed online; includes a definition of observability platforms and user reviews, 2025. [Online]. Available: https://www.gartner.com/reviews/market/observability-platforms.

[16] E. Authors, *Envoy: Open source edge and service proxy*, Accessed: 2025-08-28, 2025. [Online]. Available: https://www.envoyproxy.io/.

[17] G. Authors, *Grafana: The open and composable observability platform*, Accessed: 2025-08-28, 2025. [Online]. Available: https://grafana.com/.

[18] J. Authors, *Jaeger: Open source distributed tracing platform*, Accessed: 2025-08-28, 2025. [Online]. Available: https://www.jaegertracing.io/.

[19] K. Authors, *Kiali: The console for istio service mesh*, Accessed: 2025-08-28, 2025. [Online]. Available: https://kiali.io/.

[20] P. Authors, *Prometheus: Monitoring system time series database*, Accessed: 2025-08-28, 2025. [Online]. Available: https://prometheus.io/.

[21] A. S. Foundation, *Apache skywalking: Application performance monitoring system*, Accessed: 2025-08-28, 2025. [Online]. Available: https://skywalking.apache.org/.

[22] OpenZipkin, *Zipkin: A distributed tracing system*, Accessed: 2025-08-28, 2025. [Online]. Available: https://zipkin.io/.

[23] T. I. Authors, *Install istio with an external control plane*, https://istio.io/latest/docs/setup/install/external-controlplane/, Accessed: 2025-08-13, 2025.

[24] OWASP ModSecurity Project, *Modsecurity: Open-source web application firewall*, Official website, An open-source, cross-platform WAF engine under OWASP custodianship; accessed online, 2025. [Online]. Available: https://modsecurity.org/.

[25] Microsoft, *The stride threat model*, Accessed: 2025-08-02, 2009. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN.

[26] OWASP Foundation, *Owasp api security project*, Accessed: 2025-08-02, 2023. [Online]. Available: https://owasp.org/www-project-api-security/.

[27] A. C. S. A. .-. D. C. Training, *What is amazon api gateway?* YouTube video, Accessed via Online, 14th September 2021. [Online]. Available: https://www.youtube.com/watch?v=1XcpQHfTOvs.

[28] G. Archer, *Implementing zero trust apis*, Curity – Learn, Published on November 17, 2022; accessed online, Nov. 2022. [Online]. Available: https://curity.io/resources/learn/implementing-zero-trust-apis/.