Politecnico di Torino

Master's Degree in Cybersecurity

A.a. 2024/2025

Graduation Session December 2025

Design and Implementation of an Authentication and Authorization Framework for OpenC2

Supervisors:

Prof. Daniele Bringhenti
Prof. Matteo Repetto
Prof. Fulvio Valenza

Candidate:

Nicola Poidomani

Abstract

The Open Command and Control (OpenC2) framework is emerging as a crucial standard for orchestrating and automating defensive cyber operations, enabling interoperability between heterogeneous security tools. However, the core specifications deliberately do not mandate specific security mechanisms, creating a critical gap that could expose the command and control infrastructure to unauthorized access and malicious manipulation. This thesis addresses this gap by designing, implementing, and evaluating a comprehensive authentication and authorization framework to secure the OpenC2 ecosystem.

The proposed solution leverages industry-standard protocols to ensure robust and scalable security. For authentication and delegated access, we integrate the OAuth 2.0 framework, utilizing the Authorization Code grant flow to guarantee that only legitimate entities (Producers) can issue commands. The implementation is developed in Python, using the Authlib library to build a dedicated Authorization Server responsible for token issuance and management.

For fine-grained access control, the framework incorporates Casbin, a versatile and powerful authorization library. By enforcing policies based on the PERM (Policy, Effect, Request, Matchers) metamodel, Casbin allows the OpenC2 command recipient (Consumer) to verify whether an authenticated Producer is permitted to perform a specific action on a given target, implementing a Role-Based Access Control (RBAC) model. The entire proof-of-concept is built upon the otupy/openc2lib library, demonstrating a practical and seamless integration of these security layers into the OpenC2 message flow.

Finally, the implemented solution is validated through functional testing, which confirms the correct enforcement of access control policies, and a performance analysis, which quantifies the latency overhead introduced by the security mechanisms. The results demonstrate that the integration of OAuth 2.0 and Casbin provides a robust and viable solution for enhancing the security, trustworthiness, and operational readiness of the OpenC2 framework in real-world deployments.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter explains the context and thesis motivation. It begins by explaining the key issues with contemporary cybersecurity. It points to the increased sophistication of attacks and the necessity of defense mechanisms that can interoperate. It then introduces OpenC2 as a standard of remote command and control. The standard ensures that various security systems have consistent communications. The chapter then specifies the research goals. It centers on designing and using mechanisms of authentication and authorization. Finally, the chapter explains the thesis organization, showing how each chapter works to achieve the research intentions.

1.1 Context

The world of cybersecurity is evolving rapidly. It is characterized by sophistication and an ever-increasing amount of data. Organizations struggle with frequent attacks, including targeted attacks, sophisticated phishing attacks, ransomware, and automated intrusions. The increased sophistication has rendered traditional defense approaches, which heavily rely on human intervention and reactive postures, inefficient and impractical. Although by now, their role is not obviated and they continue to hold their relevance with regards to specific instances, they nevertheless prove too slow to compete with the velocity of contemporary attacks. They also cannot deliver the consistency that is necessary to address repeated and epidemic attacks.

To address these sorts of problems, taking up automation and security operation orchestration has now become very important. Automation enables rapid response by machines, significantly reducing the risk. Orchestration assists various tools to interoperate and execute defense strategies collaboratively. Through these strategies, security responses can be scheduled, run, and confirmed consistently, overcoming the shortcomings of isolated response.

In this context, remote control of security operations has become one of the key concepts in modern cybersecurity automation. Instead of relying on human intervention on a case-by-case basis, this approach is founded on a command and control (C2) system that allows a single authority to issue instructions to a wide range of security devices and software — including firewalls, intrusion detection and prevention systems (IDS/IPS), endpoint protection platforms, and network monitoring tools — to perform specific actions. By enabling rapid responses such as blocking a malicious IP address, isolating an infected file, or quarantining a compromised host within seconds of detection, such systems not only accelerate incident response but also reduce the likelihood of human error, thereby enhancing the overall effectiveness of defense operations.

However, remote control systems have not been effective due to limited interoperability. Security platforms and devices by various vendors employ proprietary message formats and incompatible protocols. This leads to single-boxed systems that hinder orchestration and make it impossible to have an integrated security ecosystem. To transcend these shortcomings, the Open Command and Control (OpenC2) standard was created within the OASIS organization. OpenC2 specifies a common language to control security capabilities, allowing diverse systems to exchange information uniformly and automatically [1]. Through the encouragement of interoperability between various security tools, OpenC2 enables faster, more efficient, and coordinated incident responses. The standard, therefore, can be found to act primarily as an important facilitator of contemporary cybersecurity methods by decreasing reaction times while encouraging extensible and communal defense strategies [2].

1.2 Objectives

While OpenC2 provides a standardized language for orchestrating defensive cyber operations, the core specifications deliberately do not mandate specific security mechanisms for protecting the command and control messages themselves [2]. This architectural choice offers flexibility but simultaneously introduces a critical gap: in a real-world deployment, commands such as isolating a host or blocking a malicious IP address must be protected from unauthorized access, tampering, and repudiation. Without robust security controls, an OpenC2 ecosystem could be vulnerable to hijacking by malicious actors, who could issue unauthorized commands to disable security tools or disrupt network operations.

This security gap is further amplified by the modern trend of **outsourcing cybersecurity services**, where organizations delegate the management of Security Operation Centers (SOC) and other detection, investigation, and response processes to external operators. In such scenarios, it becomes necessary to grant these third parties the ability to apply configurations or issue defensive commands to specific security appliances, but without providing them unrestricted administrative access to the entire infrastructure. This vision highlights the need for a secure delegation mechanism capable of enforcing limited, auditable, and revocable permissions. Consequently, the adoption of a framework such as **OAuth 2.0** is strongly justified: originally designed for delegated access, OAuth 2.0 offers token-based authorization, fine-grained control, and traceability, addressing the unique challenges of cross-organizational trust and control that simpler mechanisms could not adequately cover.

The main goal of this thesis is to address this gap by designing, implementing, and evaluating a comprehensive framework for *Authentication and Authorization* (AA) within an OpenC2 environment. This work aims to secure the remote control of security functions, ensuring that every command is legitimate, properly authorized, and fully traceable.

To achieve this, the following key objectives have been defined:

- **Designing a Secure Architecture:** Developing an architecture that seamlessly integrates authentication and authorization mechanisms into the OpenC2 message flow between Producers and Consumers. The proposed architecture must be robust, scalable, and compliant with established security best practices.

- **Implementing OAuth 2.0 for Authentication:** Leverage the OAuth 2.0 framework [3] as the core mechanism for authentication. This includes a detailed analysis of its various grant types (e.g., Authorization Code, Client Credentials) to identify the most suitable flow for machine-to-machine (M2M) communication, which is typical of OpenC2. The implementation will rely on the `Authlib` library within a Python ecosystem to create a dedicated Authorization Server.

- **Integrate Casbin for Fine-Grained Authorization:** Beyond authentication, implement a flexible authorization model using `Casbin`. Casbin is a powerful and versatile authorization library that supports multiple access control models, including Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC). This will enable the enforcement of granular policies, ensuring that only authorized Producers can issue specific commands to specific Consumers based on defined rules.

- **Practical Implementation and Validation:** Implement the entire framework as a proof of concept using the `otupy/openc2lib` Python library [4]. This will demonstrate the practical feasibility of the proposed solution and validate its effectiveness in a simulated environment.

Ultimately, this thesis aims to demonstrate that the integration of OAuth 2.0 and Casbin not only secures the OpenC2 framework but also enhances its trustworthiness and operational readiness for deployment in complex, multi-vendor cybersecurity environments.

1.3 Thesis Structure

This thesis is organized into the following chapters:

- **Chapter 2 – Access Control Background** introduces the fundamental concepts of access control. It reviews the main authorization models (RBAC, ABAC, and rule-based approaches) and the core authentication mechanisms (passwords, tokens, biometrics and multi-factor authentication), providing the theoretical foundation for the rest of the work.
- **Chapter 3 – The OpenC2 Framework** presents the OpenC2 standard, its Producer–Consumer architecture, and the supported transfer protocols and message formats. It details commands, responses, and actuator profiles, and discusses the security requirements that motivate the need for a dedicated authentication and authorization framework.
- **Chapter 4 – Authentication and Authorization Frameworks** describes the building blocks of the proposed solution. It introduces the OAuth 2.0 framework, its roles and grant types, and explains how access and refresh tokens are used for delegated authorization. It then presents Casbin and the PERM metamodel as a flexible policy engine for fine-grained authorization.
- **Chapter 5 – Thesis Objectives** refines the research goals and contributions of the work. It defines the design principles of the proposed architecture, maps OpenC2 components to OAuth 2.0 roles, and outlines the validation strategy based on functional testing and performance evaluation.
- **Chapter 6 – Implementation of the Authentication and Authorization Framework** details the practical realization of the architecture. It describes the overall system design, the end-to-end flow of authenticated OpenC2 commands, and the implementation of the Authorization Server, the headless User Agent, the Producer, and the Consumer, including token handling and Casbin-based policy enforcement.
- **Chapter 7 – Testing and Performance Evaluation** presents the experimental validation of the framework. It reports the functional tests used to verify correct enforcement of access control policies and the performance measurements that quantify the latency overhead introduced by OAuth 2.0

introspection and Casbin authorization, discussing their impact on OpenC2 operations.

Chapter 2

Access Control Background

This chapter provides the theoretical background necessary to understand the fundamental concepts of access control, which underpin the mechanisms of authentication and authorisation discussed in the following chapters. It introduces the main principles, models, and policies that regulate how entities gain access to resources and under what conditions such access is granted or denied. A solid understanding of these concepts is crucial for designing and implementing a secure and robust integration of authentication and authorisation within the OpenC2 framework.

Access control is a cornerstone of computer security, defining the mechanisms and policies to control access to resources. Its primary goal is to protect the confidentiality, integrity, and availability of information by ensuring only authorised entities can perform permitted actions under specific conditions [5]. Access control is a complex set of processes built upon two fundamental mechanisms:

- **Authentication**: The process of verifying the identity of an entity.
- **Authorisation**: The process of determining whether an authenticated entity has the right to access a specific resource or perform a particular action.

In the context of OpenC2, a robust implementation of authentication and authorisation is essential to ensure that only legitimate actors can issue and execute commands within a cyber defense ecosystem, preventing unauthorised access and malicious activities.

2.1 Authorisation

Authorisation is the process of defining and enforcing access rights. Access control policies are implemented through various models, each with its own strengths and weaknesses. The choice of a particular model, or combination of models, depends on the specific security requirements of a system. More advanced methods employ rule, role, or attribute-based centralized mechanisms, which provide high control and flexibility. Central to these models are often context forces like time, place, or device condition to enable dynamic and high-granular access control decisions. Modern security architectures often blend various authorisation schemes to provide strong and uniform protection to distributed infrastructures. This blended approach is particularly relevant in dynamic scenarios—cloud computing, the Internet of Things (IoT), and cyber defence architectures like OpenC2—where secure, authority-validated communication between Producers and Consuming Parties is crucial to coordinated and secure operation.

2.1.1 Role Based Access Control

Role-Based Access Control (RBAC) is a security model where access decisions are based on the **roles** that individual users have within an organization, and permissions are not assigned directly to individual users. This approach allows for more efficient, consistent, and secure management of authorizations, reducing administrative complexity and making it the most commonly used model [6].

The main components of this model are:

- **Users:** The entities (individuals or systems) that request access to system resources.
- **Roles:** Collections of permissions that represent job functions or operational responsibilities within the organization (e.g., Administrator, Analyst, Basic User).
- **Permissions (or Privileges):** The authorized operations that different roles are allowed to perform (e.g., read, write, delete).

RBAC is based on the **Principle of Least Privilege**, ensuring that each user has only the authorizations strictly necessary for their specific role. For example, within an OpenC2 environment, a 'Security Analyst' role might have permissions to query for information, while an 'Administrator' role would have the additional permissions to execute commands that modify the system's state.

2.1.2 Attribute Based Access Control

Attribute-Based Access Control (ABAC) is defined in [7] as a "logical access control methodology where authorisation to perform a set of operations is determined by evaluating attributes associated with the subject, object, requested operations, and, in some cases, environment conditions against policy, rules, or relationships that describe the allowable operations for a given set of attributes." Unlike Role-Based

Access Control (RBAC), which use roles to perform authorisation, ABAC evaluates a combination of attributes such as those related to the subject (e.g., security level, department) and the object (e.g., data classification, resource type) to determine access permissions dynamically. This allows for a more fine-grained and context-aware approach to access control, making it well-suited for complex and dynamic environments like OpenC2 [8].

2.1.3 Rule Based Access Control

Rule-Based Access Control (RuBAC) defines access decisions through explicitly defined conditional logic rules rather than relying solely on static roles or fixed attributes [9]. In contrast to other models, RuBAC is predicated on policies expressed as conditional rules that typically follow an `if-then` structure, combining various real-time and contextual variables.

These variables provide a high degree of context-awareness for access enforcement. Common variables used in RuBAC policies include:

- Time of day or day of week;
- Specific devices or mobile credentials;
- Geographic location or IP address;
- Concurrent authentication events (e.g., badge plus QR scan);
- Visitor status or pre-approved credentials.

For example, a typical RuBAC rule might be expressed as:

> *"If the request originates from the internal network* ***AND*** *the time is between 8:00 AM and 6:00 PM,* ***THEN*** *allow access to the resource."*

This approach allows for the definition of more granular and adaptable policies compared to traditional models. In practice, RuBAC is often implemented as a layer of conditions on top of Role-Based Access Control (RBAC), providing the foundational structure while RuBAC adds the necessary contextual constraints for a more dynamic and secure authorization process. The combination of RBAC and RuBAC is a powerful strategy that can provide both the structure of role-based permissions and the flexibility of rule-based conditions.

2.2 Authentication

Authentication is the process of verifying the declared identity of an entity, such as a user, a device, or a process, before granting it access to a protected resource.

Authentication answers the question: *"Who are you?"* and ensures that the entity presenting credentials genuinely corresponds to that identity. Once authentication is completed, the system can determine what operations the entity is authorized to perform [5].

Authentication can be achieved through several mechanisms, typically categorized by the type of authentication factor used [10]:

- **Knowledge-based factors**: Something the user knows, such as a password or a Personal Identification Number (PIN).
- **Possession-based factors**: Something the user has, such as a hardware token, a smart card, or a mobile device generating One-Time Passwords (OTP) or holding cryptographic credentials.
- **Inherence-based factors**: Something the user *is* or *does*, for instance, biometric features (fingerprint, face, iris) or behavioral traits such as keystroke dynamics or gait recognition.
- **Location-based factors**: Where the user is, such as their geographical location or IP address.
- **Behavioral-based factors**: How the user acts, such as their typing speed or mouse movements.

To increase reliability, modern systems often employ **Multi-Factor Authentication (MFA)**, which combines two or more distinct factors (e.g., knowledge and possession) to significantly reduce the risk associated with the compromise of any single factor [5].

2.2.1 Password Authentication

Password-based authentication is the most traditional and widely adopted method for verifying a user's identity. It relies on a **shared secret**, typically a string known only to the user and the authentication system, that must be presented correctly to gain access [5].

Despite its simplicity and ubiquitous use, this method is vulnerable to several threats, including:

- **Brute-force attacks**: An attacker attempts to guess a password by systematically trying every possible combination of characters.
- **Dictionary attacks**: A more targeted form of brute-force attack where the attacker uses a list of common words and phrases.

- **Credential stuffing**: Attackers use credentials stolen from one service to attempt to log in to another service.
- **Phishing**: Attackers trick users into revealing their credentials through deceptive emails or websites [11].

For this reason, modern systems rarely rely on passwords alone, instead reinforcing them with additional measures such as password hashing, salting, and rate-limiting mechanisms to mitigate brute-force attacks.

2.2.2 Token Authentication

Token-based authentication represents a more modern and secure approach compared to traditional password mechanisms. Instead of repeatedly transmitting user credentials, the system issues a cryptographically protected token after a successful authentication phase. This token acts as a temporary proof of identity and authorisation, allowing the user or client to access protected resources without resubmitting sensitive credentials [5, 3].

A popular format for tokens is the **JSON Web Token (JWT)**, an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object [12]. A JWT consists of three parts separated by dots (‘.’):

- **Header**: Typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
- **Payload**: Contains the claims, which are statements about an entity (typically, the user) and additional data.
- **Signature**: Used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn’t changed along the way.

Tokens are typically short-lived and contain information such as the user identity, granted permissions (scope), and expiration time. They can be opaque strings or structured data formats such as JWTs, which include cryptographically signed claims to ensure integrity and authenticity. This design reduces the exposure of credentials, limits the duration of access, and supports stateless communication between distributed components.

Token-based authentication plays a central role in modern web and distributed architectures, where clients and services operate independently. Its relevance is particularly evident in the OAuth 2.0 framework [3], which relies on access tokens as the primary mechanism for authenticating clients and delegating user access. Through this model, authentication is decoupled from authorisation, allowing secure delegation of rights without exposing user credentials, a concept that will be explored in more detail in the following sections.

Token authentication enhances both security and scalability:

- **Security:** tokens can be revoked, refreshed, or scoped to specific actions or resources, reducing the impact of compromise [13].
- **Scalability:** tokens allow stateless verification by resource servers, avoiding persistent sessions and enabling distributed deployments.

However, token management introduces its own challenges, such as secure storage on the client side, proper handling of token expiration and renewal, and the need to protect against replay or interception attacks. For this reason, token transmission is generally performed over encrypted channels (e.g., HTTPS/TLS), and refresh tokens are used to renew expired access credentials in a controlled way [3, 5].

2.2.3 Biometric Authentication

Biometric authentication relies on unique physiological or behavioural traits of individuals to verify their identity. Rather than something the user knows or has, biometrics authenticate based on something the user *is* or *does* [5]. Common modalities include fingerprints, facial features, iris/retina scans, voice, as well as behavioural patterns such as typing rhythm, gait, or touch dynamics.

Biometric methods are often praised for their usability and resilience against credential theft. Since biometric traits are inherently tied to the individual, they cannot be easily shared or forgotten, reducing typical user burdens associated with passwords or tokens [14]. However, the use of biometric data raises significant privacy and security concerns. Unlike passwords, biometric identifiers cannot be changed if they are compromised. A data breach that exposes a database of fingerprints or facial scans could have long-lasting consequences for the affected individuals [15]. Furthermore, biometric systems can be vulnerable to "spoofing" attacks, where an attacker uses a fake biometric sample (e.g., a photograph or a latex fingerprint) to fool a sensor.

In practice, biometric authentication is often used in conjunction with other factors (e.g. in multi-factor schemes) to balance security and usability.

Chapter 3

The openc2 Framework

3.1 Introduction

Open Command and Control (OpenC2) is a standardized, machine-readable language designed to orchestrate and automate cybersecurity operations in real-time. In an ecosystem where security tools from different vendors rarely interoperate seamlessly, OpenC2 provides a common language that enables disparate technologies, such as firewalls, intrusion detection systems (IDS), and endpoint detection and response (EDR) solutions, to work in concert [2]. This interoperability is crucial for moving beyond slow, manual response processes and toward coordinated, automated defense.

Developed under the stewardship of the OASIS standards organization, OpenC2 aims to reduce response times from minutes or hours to machine speed. The fundamental premise is that if security components can understand each other's commands, an entire defense infrastructure can react to a threat as a single, cohesive unit. This is achieved by abstracting cyber defense actions into a standard set of **Actions** performed on well-defined **Targets**.

However, a language for command and control inherently introduces significant security risks. If a malicious actor gains the ability to issue OpenC2 commands, they could systematically disable an organization's defenses or manipulate them for malicious purposes. The OpenC2 standard anticipates this, specifying that commands must be transmitted over a secure and authenticated channel, but it intentionally does not mandate a single, universal mechanism for achieving this security. This deliberate gap creates the central challenge addressed by this thesis: designing a robust and scalable framework for authentication and authorisation tailored to the OpenC2 ecosystem, using modern standards such as OAuth2 and Casbin.

3.2 Core Architecture

The OpenC2 architecture is defined by a simple yet powerful *request-response* paradigm between two primary components: the **Producer** and the **Consumer**. This model decouples the decision-making logic from the execution mechanics, fostering a flexible and scalable environment for cyber defense orchestration [2].

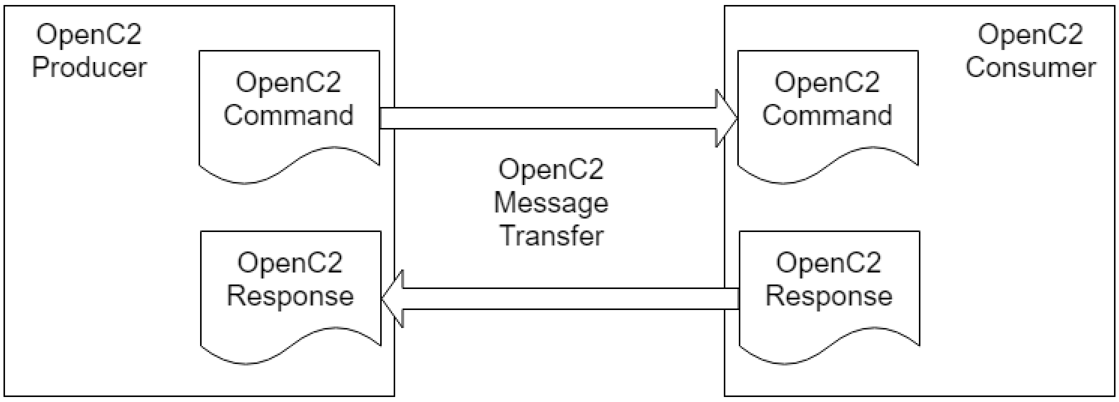


Figure 3.1: The OpenC2 Producer-Consumer Model [2]

3.2.1 Architectural Roles

Producer A Producer is an entity that generates and issues commands. This component typically houses the higher-level logic for cybersecurity management, such as a Security Orchestration, Automation, and Response (SOAR) platform. It determines *what* action needs to be taken in response to a threat and *which* target is affected. As defined in the specification:

> "A Producer is an entity that creates and transmits Commands instructing one or more systems to execute Actions as specified in the Command." [2]

Consumer A Consumer is the entity that receives, parses, and acts upon a command. It is the component responsible for executing the requested action. A Consumer may be a security sensor, a network device, or any other managed cyber defense tool. The specification defines it as:

> "An entity that receives and may act upon a Command. A Consumer may create Responses that provide any information captured or necessary to send back to the Producer." [2]

Within a Consumer, one or more **Actuators** are responsible for translating the abstract OpenC2 command into the specific, proprietary instructions required by the underlying technology (e.g., changing a firewall rule or quarantining a file on an endpoint).

This clear separation of roles is the foundation of OpenC2's vendor-agnostic and interoperable design. It allows a single Producer to control a heterogeneous collection of security tools without needing to understand the specific implementation details of each one.

3.2.2 Transfer Protocols and Message Payloads

The OpenC2 architecture is transport-agnostic, meaning it does not mandate a single communication protocol. This flexibility allows it to be adapted to a wide variety of environments. However, to ensure interoperability, the OASIS technical committee has standardized specifications for common protocols. These include:

- **HTTPS**: For request-response interactions over standard web infrastructure, secured with Transport Layer Security (TLS) [1].
- **MQTT**: For publish-subscribe messaging patterns, often used in IoT and distributed environments where low-overhead, asynchronous communication is beneficial.

Regardless of the transfer protocol, the message payload must be serialized in a mutually understood format. The primary serialization format for OpenC2 messages is **JSON** (JavaScript Object Notation), due to its human-readability and widespread support across programming languages. For environments where bandwidth and processing power are constrained, **CBOR** (Concise Binary Object Representation) is specified as a more compact binary alternative.

3.2.3 Commands and Responses

Communication in OpenC2 is structured around two message types: the **Command** and the **Response**.

An OpenC2 Command is a structured message that instructs a Consumer to perform a specific task. According to the *OpenC2 Language Specification*, every command consists of two mandatory fields and several optional ones [1]:

- **Action (mandatory)**: A verb defining the operation to be performed (e.g., `deny`, `allow`, `query`).
- **Target (mandatory)**: The object of the action, specifying what the action is to be performed on (e.g., an `ipv4_net`, a `domain_name`, or a `file`).
- **Arguments (optional)**: Modifiers that provide context for the command, such as timing (`start_time`, `duration`) or requesting a response (`response_requested`).

- **Actuator (optional)**: Specifies which sub-component of the Consumer should execute the command, which is essential when a Consumer manages multiple functions (e.g., a firewall and an IDS).
- **Command ID (optional but recommended)**: A unique identifier for tracking and correlating commands with their corresponding responses.

The **Response** is a message sent from the Consumer back to the Producer to acknowledge receipt and indicate the outcome of the command execution. A response message includes:

- **Status**: A numerical status code indicating the result (e.g., 102 for Processing, 200 for OK, 400 for Bad Request, 401 for Unauthorized, 501 for Not Implemented).
- **Status Text (optional)**: A human-readable string further describing the status.
- **Results (optional)**: A dictionary containing any data returned by the command execution, such as the output of a `query` action.

The following example, drawn from the SLPF profile, illustrates this interaction:

```
Command:
{
  "action": "allow",
  "target": {
    "ipv6_connection": {
      "protocol": "tcp",
      "dst_addr": "3ffe:1900:4545:3::f8ff:fe21:67cf",
      "src_port": 21
    }
  },
  "actuator": {
    "slpf": {}
  }
}

Response:
{
  "status": 200,
  "results": {
    "slpf": {
      "rule_number": 1234
    }
  }
}
```

In this example, the Producer sends a command to allow FTP connections to a specific IPv6 address. The Consumer applies the corresponding filtering rule and returns a response confirming successful execution, along with the identifier of the newly created rule.

3.2.4 Actuator Profiles

While the OpenC2 language defines a set of common Actions and Targets, it cannot possibly cover every capability of every security product. To address this, the standard uses **Actuator Profiles**. A profile is a separate specification that defines the specific vocabulary applicable to a particular cyber defense function, such as stateless packet filtering, endpoint security, or threat intelligence [16].

Profiles serve two key purposes:

1. **Standardization**: They define the precise combinations of actions, targets, and arguments that are valid for a given function. For example, the SLPF profile specifies that a firewall can `deny` an `ipv4_net` but does not define an action to `reboot` it.

2. **Extensibility**: They allow the OpenC2 language to be extended in a controlled and interoperable manner. New technologies can be integrated into the OpenC2 ecosystem by creating a new profile that defines their unique capabilities.

By using profiles, OpenC2 maintains a balance between a standardized core language and the flexibility required to support a wide and ever-evolving range of security technologies.

3.3 Security Considerations and Requirements

The power to command and control an organization's entire security infrastructure from a central point makes the OpenC2 ecosystem a high-value target for attackers. The OpenC2 specifications acknowledge this and mandate the use of secure transport protocols to protect the confidentiality and integrity of messages in transit. For instance, the HTTPS transfer specification recommends the use of TLS for encrypted communication [17].

However, transport-level security alone is insufficient. A comprehensive security solution for OpenC2 must address two higher-level challenges:

- **Authentication**: How does a Consumer verify that a command originates from a legitimate, trusted Producer? How can a Producer trust the responses from a Consumer? Without strong authentication, a malicious actor could impersonate a Producer and inject unauthorized commands into the system.
- **Authorisation**: Once a Producer is authenticated, how does a Consumer determine if that specific Producer is permitted to issue a particular command? A Producer responsible for monitoring might be authorized to issue `query` commands but should be denied from executing disruptive `deny` or `delete` commands.

These requirements highlight the need for a robust access control framework that can be integrated seamlessly into the OpenC2 architecture. This framework must be capable of managing complex relationships between multiple Producers and Consumers, enforcing granular policies, and operating effectively in a distributed, heterogeneous environment. The following chapters will explore how the OAuth 2.0 framework and the Casbin authorisation library can be combined to meet these critical security needs.

Chapter 4

Authentication and Authorization Frameworks

4.1 Introduction

The goal of this thesis is to ensure that OpenC2 commands are both **authenticated** and **authorized**. In the *Access Control* chapter, authentication (*who you are*) and authorization (*what you can do*) were clearly distinguished, emphasizing that in distributed systems they must be treated as separate yet coordinated functions: authentication verifies identity or attributes, whereas authorization determines the corresponding permissions according to defined policies and contextual constraints. This distinction also guides the proposed architecture: **OAuth 2.0** is adopted for *delegation* and *token management* (who can access and to what extent), whereas a policy engine such as **Casbin** is responsible for the *fine-grained enforcement* of access control decisions (what actions are permitted on OpenC2 commands).

This chapter establishes the theoretical background for this two-part solution. It is structured to first address the challenge of authentication and delegation, followed by the challenge of fine-grained authorization.

For the first part, we introduce the **OAuth 2.0** framework, referring to the official **RFC 6749** specification [3] as the normative basis. This framework is motivated by its role as the industry standard for managing delegated access and issuing secure access tokens. This mechanism is fundamental as it allows a client (the OpenC2 Producer) to prove its identity to a resource server (the Consumer) without sharing primary credentials.

For the second part, we introduce **Casbin**. We will describe its flexible, model-driven architecture, centered on the PERM (Policy, Effect, Request, Matchers) metamodel. Casbin is chosen for its ability to consume the authenticated identity provided by OAuth 2.0 and use it to enforce granular, policy-based decisions on the incoming OpenC2 commands.

Together, these sections provide the necessary background to understand the design and practical implementation of the complete security framework, which is detailed in the following chapter.

4.2 OAuth 2.0 Framework Overview

OAuth 2.0 is an *authorization framework* that enables an application (*client*) to obtain *tokens* with limited privileges to access protected resources, avoiding the need to share the resource owner's original credentials (*resource owner*). Compared to OAuth 1.0 [18], OAuth 2.0 [3] simplifies the model, makes roles explicit, and broadens support for *public clients*. It also shifts protection from per-request *request signing* (typical of OAuth 1.0) to systematic use of *TLS* and *Bearer Tokens* [19]. Recent best practices recommend additional hardening (e.g., PKCE for public clients, deprecation of the Implicit flow, tighter *scope* and token TTL) [20].

4.2.1 Roles

The framework defines four fundamental roles [3]:

- **Resource Owner (RO)**: the party that controls the resource (a person or, in Machine-to-Machine (M2M) scenarios, a system entity).
- **Client**: the application requesting access. It may be *confidential*, able to securely store credentials (e.g., backend servers), or *public*, unable to protect them (e.g., single-page or mobile applications).
- **Authorization Server (AS)**: issues tokens after obtaining/evaluating the RO's consent.
- **Resource Server (RS)**: exposes the API/resource and validates tokens presented by the client.

4.2.2 Protocol Flow

The abstract protocol flow, as defined in RFC 6749 [3], describes the interaction between the main roles of the OAuth 2.0 framework. The process is divided into two primary phases: obtaining an authorization grant and exchanging it for an access token.

1. **Authorization Request:** The client requests authorization from the Resource Owner, typically by redirecting the user agent (e.g., a web browser) to the Authorization Server. This interaction occurs through the *front-channel*.

2. **Authorization Grant:** The Resource Owner authenticates with the Authorization Server and grants the client's request. The Authorization Server then redirects the user agent back to the client with an authorization grant.

3. **Access Token Request:** The client exchanges the authorization grant for an access token by sending a direct `POST` request to the Authorization Server's token endpoint. This step occurs via the *back-channel*, invisible to the user agent.

4. **Access Token Response:** The Authorization Server authenticates the client, validates the authorization grant, and, if valid, issues an access token.

5. **Accessing the Protected Resource:** The client presents the access token to the Resource Server when requesting access to a protected resource.

6. **Resource Served:** The Resource Server validates the token and, if it is valid and authorized, returns the requested resource to the client.

This separation between the *front-channel* (browser-based redirection) and the *back-channel* (direct server-to-server communication) is fundamental to the OAuth 2.0 security model, ensuring that sensitive credentials and tokens are exchanged securely.

```
+--------+                               +---------------+
|        |--(A)- Authorization Request ->|   Resource    |
|        |                               |     Owner     |
|        |<-(B)-- Authorization Grant ---|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(C)-- Authorization Grant -->| Authorization |
| Client |                               |     Server    |
|        |<-(D)----- Access Token -------|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(E)----- Access Token ------>|    Resource   |
|        |                               |     Server    |
|        |<-(F)--- Protected Resource ---|               |
+--------+                               +---------------+
```

Figure 4.1: Abstract Protocol Flow

4.2.3 Authorization Grants, Access Tokens, and Refresh Tokens

OAuth 2.0 defines several core artifacts that enable delegated authorization and controlled access to protected resources:

- **Authorization Grant**: A temporary credential representing the Resource Owner's consent. It is exchanged by the client at the Authorization Server's *token endpoint* to obtain an access token. Different grant types (e.g., *authorization code*, *client credentials*) define distinct ways to obtain and use this credential, each with specific security assumptions and applicability conditions [3].
- **Access Token**: A credential representing an authorization with a defined *scope* and limited *lifetime*. It allows the client to access protected resources and is typically used as a *Bearer* token in the HTTP header: `Authorization: Bearer <token>`. Because any party in possession of a bearer token can use it, tokens must be transmitted exclusively over secure channels (TLS) and stored securely [19].
- **Refresh Token**: An optional credential that allows the client to obtain new access tokens without re-prompting the Resource Owner. It is usually issued only to confidential or otherwise trusted clients and should be managed under strict rotation and revocation policies [3, 20].

4.3 OAuth 2.0 Grant types

The framework specifies several authorization grants, or "flows," for obtaining an access token. The choice of grant type is critical for security and depends on the client's type (confidential or public) and the specific use case.

4.3.1 Authorization Code Grant

The Authorization Code grant is the most commonly used and most secure OAuth 2.0 flow. It is designed for **confidential clients** (e.g., traditional web applications with a server-side backend) capable of securely storing a client secret.

Flow Description

As defined in RFC 6749 [3], the process consists of the following steps:

1. **Client initiates the authorization request:** The client application directs the user's user agent (e.g., a web browser) to the Authorization Server's `/authorize` endpoint. The request includes several key parameters:

    - `response_type=code`: Indicates that the client requests an authorization code.
    - `client_id`: The client's unique identifier assigned by the Authorization Server.
    - `redirect_uri`: The URI to which the Authorization Server will redirect the user after authorization.
    - `scope`: The permissions being requested by the client.
    - `state`: A value used to prevent Cross-Site Request Forgery (CSRF) attacks. The Authorization Server returns this value unchanged.

2. **User authenticates and grants consent:** The Authorization Server prompts the Resource Owner to log in and approve or deny the client's request.

3. **Authorization Server redirects with authorization code:** If access is granted, the Authorization Server redirects the user agent to the client's registered `redirect_uri`, including the temporary `code` and the `state` value in the query string.

4. **Client exchanges the code for tokens:** The client application receives the redirect and makes a direct (back-channel) `POST` request to the Authorization Server's `/token` endpoint. The request includes:

    - `grant_type=authorization_code`: Specifies the type of grant.
    - `code`: The authorization code obtained in the previous step.
    - `redirect_uri`: Must exactly match the URI used in the initial request.
    - `client_id` and `client_secret`: The client's credentials for authentication.

5. **Authorization Server issues tokens:** The Authorization Server validates the client credentials and authorization code. If valid, it issues an access token and, optionally, a refresh token, which the client can use to access protected resources.

```
+----------+
| Resource |
|   Owner  |
|          |
+----------+
     ^
     |
    (B)
+----|-----+          Client Identifier      +---------------+
|         -+----(A)-- & Redirection URI ---->|               |
|  User-   |                                 | Authorization |
|  Agent  -+----(B)-- User authenticates --->|     Server    |
|          |                                 |               |
|         -+----(C)-- Authorization Code ---<|               |
+-|----|---+                                 +---------------+
  |    |                                         ^      v
 (A)  (C)                                        |      |
  |    |                                         |      |
  ^    v                                         |      |
+---------+                                      |      |
|         |>---(D)-- Authorization Code ---------'      |
|  Client |          & Redirection URI                  |
|         |                                             |
|         |<---(E)----- Access Token -------------------'
+---------+       (w/ Optional Refresh Token)
```

Figure 4.2: Authorization Code Grant flow [3].

Security Considerations

This flow is considered robust because the access token is transmitted exclusively through a secure back-channel between the client and the Authorization Server and is never exposed to the user agent. This design mitigates threats such as token leakage through browser history or Cross-Site Scripting (XSS) attacks.

For **public clients** (e.g., mobile or single-page applications), this flow is strengthened using the **Proof Key for Code Exchange (PKCE)** extension [21], which prevents authorization code interception attacks and is now recommended for all OAuth 2.0 clients.

4.3.2 Implicit Grant (Deprecated)

The Implicit grant was originally defined in RFC 6749 [3] for **public clients** such as single-page or JavaScript-based applications that could not securely store a client secret. However, it is now considered obsolete and **strongly discouraged** by current security best practices [22], due to its inherent exposure of tokens within the user agent.

Flow Description

The Implicit flow is simplified compared to the Authorization Code grant and takes place entirely within the user agent. It proceeds as follows:

1. **Client initiates the request:** The client redirects the user's user agent (e.g., a web browser) to the Authorization Server's `/authorize` endpoint, including:

   - `response_type=token`: Indicates that the client requests an access token directly, rather than an authorization code.
   - `client_id`: The unique identifier of the client application.
   - `redirect_uri`: The URI to which the Authorization Server will redirect the user after authorization.
   - `scope`: (Optional) The requested permissions.
   - `state`: A random value used to maintain state between request and callback and to protect against Cross-Site Request Forgery (CSRF) attacks.

2. **User authenticates and grants consent:** The Authorization Server prompts the Resource Owner to authenticate and approve the client's request.

3. **Authorization Server redirects with token:** Upon successful authorization, the Authorization Server redirects the user agent to the client's `redirect_uri`, embedding the access token in the URI fragment, e.g.: `https://client.example.com/callback#access_token=ABC123&state=XYZ`

4. **Client extracts the token:** The client-side script running in the browser parses the URI fragment to retrieve the access token, which is then used in API requests to the Resource Server.

```
+----------+
| Resource |
|  Owner   |
|          |
+----------+
     ^
     |
    (B)
+----|-----+          Client Identifier     +---------------+
|         -+----(A)-- & Redirection URI --->|               |
|  User-   |                                | Authorization |
|  Agent  -|----(B)-- User authenticates -->|     Server    |
|          |                                |               |
|          |<---(C)--- Redirection URI ----<|               |
|          |          with Access Token     +---------------+
|          |            in Fragment
|          |                                +---------------+
|          |----(D)--- Redirection URI ---->|   Web-Hosted  |
|          |          without Fragment      |     Client    |
|          |                                |    Resource   |
|     (F)  |<---(E)------- Script ---------<|               |
|          |                                +---------------+
+-|--------+
  |    |
 (A)  (G) Access Token
  |    |
  ^    v
+---------+
|         |
|  Client |
|         |
+---------+
```

Figure 4.3: Implicit Grant flow [3].

Security and Deprecation Rationale

The Implicit grant has been deprecated due to several structural security weaknesses:

- **Token Exposure:** The access token is returned directly to the user agent, making it susceptible to theft through Cross-Site Scripting (XSS), browser history leaks, or malicious extensions.
- **Lack of Refresh Tokens:** Since this flow does not issue refresh tokens, the client must repeatedly obtain new access tokens, degrading both security and user experience.
- **No Client Authentication:** The client cannot be reliably authenticated, which increases the risk of impersonation and token misuse.

4.3.3 Resource Owner Password Credentials Grant (Discouraged)

The Resource Owner Password Credentials (ROPC) grant allows a client to obtain an access token by directly handling the Resource Owner's username and password. Its use is **strongly discouraged** and should be limited to highly trusted, first-party applications where redirect-based flows are not feasible.

Flow Description

As defined in RFC 6749 [3], this flow involves a direct exchange of the user's credentials for an access token:

1. **User provides credentials to the client:** The Resource Owner enters their username and password directly into the client application.

2. **Client requests a token from the Authorization Server:** The client sends a `POST` request to the Authorization Server's `/token` endpoint, including:

   - `grant_type=password`: Specifies the grant type.
   - `username`: The Resource Owner's username.
   - `password`: The Resource Owner's password.
   - `scope`: (Optional) The requested access scope.

   The client authenticates using its own credentials (`client_id` and `client_secret`), typically through the HTTP `Authorization` header.

3. **Authorization Server issues tokens:** The Authorization Server validates both the client's and the user's credentials. If successful, it returns an access token and, optionally, a refresh token.

```
+----------+
| Resource |
|  Owner   |
|          |
+----------+
     v
     |    Resource Owner
    (A) Password Credentials
     |
     v
+---------+                                  +---------------+
|         |>--(B)---- Resource Owner ------->|               |
|         |         Password Credentials     | Authorization |
| Client  |                                  |     Server    |
|         |<--(C)---- Access Token ---------<|               |
|         |    (w/ Optional Refresh Token)   |               |
+---------+                                  +---------------+
```

Figure 4.4: Resource Owner Password Credentials Grant flow [3].

Security Considerations

This grant type violates the fundamental principle of OAuth 2.0 delegated authorization without credential sharing. By requiring the Resource Owner to disclose their username and password directly to the client, it exposes sensitive credentials and increases the attack surface. Moreover, it conditions users to trust arbitrary applications with their credentials, encouraging unsafe habits and facilitating phishing attacks.

4.3.4 Client Credentials Grant

The Client Credentials grant is used for non-interactive, machine-to-machine (M2M) communication, where the client acts on its own behalf rather than on behalf of a Resource Owner. It is commonly employed by backend services, daemons, or APIs that need to authenticate and authorize themselves to access protected resources.

Flow Description

As defined in RFC 6749 [3], this flow consists of a direct exchange between the client and the Authorization Server, without any user interaction:

1. **Client requests an access token:** The client sends a `POST` request to the Authorization Server's `/token` endpoint. The request body includes:
   - `grant_type=client_credentials`: Indicates the grant type.

- `scope`: (Optional) Specifies the access scope requested by the client.

The client authenticates using its `client_id` and `client_secret`, typically through the HTTP `Authorization` header with Basic authentication.

2. **Authorization Server issues an access token:** The Authorization Server validates the client's credentials and, if successful, returns an access token representing the client's own authorization context. Refresh tokens are generally not issued in this flow, as the client can request a new access token whenever necessary.

```
+---------+                                  +---------------+
|         |                                  |               |
|         |>--(A)- Client Authentication --->| Authorization |
| Client  |                                  |     Server    |
|         |<--(B)---- Access Token ---------<|               |
|         |                                  |               |
+---------+                                  +---------------+
```

Figure 4.5: Client Credentials Grant flow [3].

Security Considerations

The Client Credentials grant is considered secure when properly implemented, since no user credentials are involved. However, the following measures are essential to ensure its security:

- **Protect client credentials:** The `client_secret` must be securely stored and transmitted only over HTTPS. It should never be embedded in public or client-side code.

- **Limit token scope:** Access tokens should be issued with the minimum privileges necessary to reduce potential impact if compromised.

- **Use strong client authentication:** When possible, prefer mutual TLS (mTLS) or JWT-based client authentication (`private_key_jwt`) instead of static shared secrets.

- **No user context:** Since no Resource Owner is involved, access control decisions must rely solely on the client's identity and assigned privileges.

4.4 From Authentication to Fine-Grained Authorization

As discussed earlier in this chapter, OAuth 2.0 provides a standardized framework for authentication and delegation, allowing a *Producer* to prove its identity and demonstrate the right to interact with a *Consumer*. While this mechanism effectively answers the question *"who are you?"*, it does not, by itself, define *"what are you allowed to do?"*.

In a distributed control environment such as OpenC2, this distinction becomes essential. Authenticating the source of a command is only the first step toward ensuring secure coordination between Producers and Consumers. It is equally necessary to determine, with fine-grained precision, whether a received command is authorized to perform the requested action on its intended target. This verification process evaluating both the command's origin and the operation it seeks to execute constitutes the foundation of authorization.

To address this requirement, this work integrates **Casbin**[1], an open-source, policy-based authorization engine, into the OpenC2 Consumer. Casbin provides a flexible, model-driven approach that decouples authorization logic from application code, making it particularly suitable for modular and distributed architectures. It supports several access control paradigms, including Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC), as described in Chapter 2. Supported by a robust Python implementation and formal model definitions [23, 24], Casbin offers a solid foundation for fine-grained, transparent, and interoperable authorization within the OpenC2 framework.

4.5 Casbin Framework Overview

Casbin is a versatile authorization framework architected on the principle of decoupling its core components: the *enforcement logic*, the *policy definitions*, and the *storage layer* [23]. This separation of concerns is fundamental, as it allows authorization rules to be managed independently of the application source code. The entire framework is unified under the **PERM** metamodel, which abstracts any access control decision into four key elements: **P**olicy, **E**ffect, **R**equest, and **M**atchers.

The operational heart of the framework is the **Enforcer**. This engine is responsible for evaluating an incoming authorization request—typically expressed as a tuple `(subject, object, action)`—and returning a binary decision (*allow* or

[1]`https://casbin.org`

deny). To do so, the Enforcer dynamically loads two distinct artifacts:

The Model Definition A configuration file (e.g., `model.conf`) that formally defines the structure of the request, the syntax of the policies, and the matching logic that binds them. It is this externalized model that makes Casbin *model-agnostic*, capable of implementing various access control schemes like RBAC or ABAC.

The Policy Set The concrete access rules, which are persisted separately from the model. Policies can be stored in a simple file (e.g., a `.csv` file) or managed via an **Adapter** connected to a database or other storage backend.

This modular design ensures that Casbin can be deployed in diverse environments, from monolithic services to distributed systems like OpenC2.

Furthermore, Casbin provides several advanced features critical for modern security architectures:

- **Role Hierarchies:** Support for permission inheritance and multi-level delegation.
- **Domain/Tenant Separation:** Policy isolation for multi-tenant applications.
- **Custom Matcher Functions:** Extensible logic using functions like `ipMatch` for IP-based filtering or `keyMatch2` for path-based control.
- **Policy-Effect Strategies:** Defines how multiple matching policies are reconciled (e.g., `allow-override`, `deny-override`).

4.6 The PERM Metamodel in Detail

As introduced, Casbin's flexibility stems from its use of the PERM (Policy, Effect, Request, Matchers) metamodel, which allows developers to formally define a wide range of access control models in a single configuration file [25]. Each section of the model plays a precise role in the authorization process:

`[request_definition]` This section defines the structure of an incoming access request. It specifies the names and the order of the parameters that will be passed to the `enforce()` method. The most common definition is a tuple of subject, object, and action:

```
[request_definition]
r = sub, obj, act
```

[policy_definition] Here, the structure of a single policy rule is defined. The definition specifies the fields that compose a policy, mapping them to the elements of the request. For a standard RBAC model, a policy links a subject (or role), an object, and an action:

```
[policy_definition]
p = sub, obj, act
```

[policy_effect] This crucial section determines how the results of multiple matching policies are consolidated into a single decision. The most common effect strategy is `allow-override`, which grants access if at least one policy permits it. This is expressed as:

```
[policy_effect]
e = some(where (p.eft == allow))
```

A more restrictive strategy, `deny-override`, could be implemented by ensuring that at least one policy allows the request and none deny it: `e = some(where (p.eft == allow)) && !some(where (p.eft == deny))`.

[matchers] This is where the core matching logic resides. The matcher is an expression that evaluates to true if an incoming request `r` matches a policy rule `p`. For RBAC, this involves checking for role inheritance using the `g()` function, alongside object and action equality:

```
[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
```

In addition to these, models implementing RBAC must include a `[role_definition]` section to define the structure of role inheritance (e.g., `g = _, _` signifies that a user can be assigned to a role).

Chapter 5

Thesis Objectives

OpenC2 standardizes the language for cyber-defense commands while leaving authentication and authorization to the implementer. This maximizes flexibility across deployments but opens a concrete security gap: there is no standard, interoperable way to prove the identity of a Producer and to enforce fine-grained policy at the Consumer. This thesis closes that gap with a standards-based Authentication and Authorization (AA) framework that is *native* to OpenC2 and compatible with existing transfer bindings.

In practical terms, the work pursues four objectives:

- integrate an OAuth 2.0–based identity layer into OpenC2 without altering the language or transfer bindings, preserving interoperability;
- employ Casbin to enforce fine-grained, auditable authorization over `action`/`target` pairs with predictable correctness;
- quantify the performance overhead introduced by per-request token introspection and policy evaluation, from both Producer and Consumer perspectives;
- ensure portability and maintainability through centralized policy with distributed enforcement, remaining compatible with off-the-shelf Authorization Servers.

The central contribution is a cohesive architecture that unifies OAuth 2.0 (identity and delegation) with Casbin (policy enforcement) inside the OpenC2 ecosystem. The design is guided by three principles. First, *separation of concerns*: authentication (who acts) and authorization (what is allowed) remain distinct, tokens carry identity and claims; policies decide on actions and targets *after* token validation. Second, *standards-based interoperability*: open interfaces avoid lock-in and respect OpenC2's separation of language and transfer. Third, *centralized policy* *with distributed enforcement*: policies are authored centrally but enforced at the edge, directly on the Consumer, in line with OpenC2's operating model.

These principles translate into a clear role mapping. The OpenC2 *Producer* acts as the OAuth 2.0 *Client*; the OpenC2 *Consumer* is both the OAuth 2.0 *Resource Server* and the local *Policy Enforcement Point (PEP)*; the human *Operator* corresponds to the *Resource Owner*; and a dedicated *Authorization Server (AS)* issues and validates tokens as the root of trust. At runtime, the Producer sends bearer-authenticated OpenC2 requests over the standard HTTPS binding. The Consumer extracts the token, performs an introspection call to the AS to verify liveness and claims, and then evaluates the requested `action/target` through a local Casbin enforcer. Only when both checks succeed is the command dispatched to the actuator, with logs and metrics making the decision path auditable.

A key challenge is reconciling the browser-oriented Authorization Code flow with console-driven Producers. To keep the same security properties without forcing interactive browsing, the design introduces a headless *User Agent (UA)* that programmatically follows redirects, submits login forms, and processes consent on behalf of the Operator. In this way, a CLI Producer can obtain short-lived access tokens securely while remaining compliant with the OpenC2 HTTPS/TLS binding. On the receiving side, the Consumer enforces a two-step pipeline, *introspection* followed by *policy evaluation*, which cleanly decouples the global lifecycle of identities and claims (anchored at the AS) from local, explainable allow/deny decisions at the edge.

The scope assumes a controlled environment: Producer, UA, Consumer, and AS communicate over HTTP; tokens are short-lived; and policies are explicit and versioned. The AS is trusted to issue and validate tokens; clocks are reasonably synchronized; the Consumer can reach the AS introspection endpoint; and the UA acts as an honest proxy for the Operator. Token replay windows and revocation latency are mitigated by short token lifetimes and introspection, although they are not exhaustively modeled.

Validation follows two complementary axes. First, *functional correctness* checks that the Consumer authorizes only policy-compliant `action/target` pairs under valid tokens, covering both positive and negative cases. Second, *performance analysis* quantifies the overhead introduced by per-request introspection and Casbin evaluation, using Producer round-trip time and Consumer processing times, and supports the attribution with packet-level traces. Together, these experiments establish both correctness and the expected cost of strong, per-request validation.

In summary, the thesis contributes: a standards-aligned AA architecture for OpenC2; a precise mapping of roles and interactions; a headless UA to enable secure CLI flows; and a Consumer-side validation pipeline that is both auditable and interoperable. The next chapter details the implementation of the *Producer*, the *Authorization Server* integration, the *User Agent*, and the *Consumer*, and shows how the `otupy` library parses and serializes OpenC2 messages while remaining faithful to OpenC2 semantics.

Chapter 6

Implementation of the Authentication and Authorization Framework

6.1 Introduction

The primary objective of this chapter is to describe the practical implementation of the **Authentication and Authorization framework** within the OpenC2 environment. This section transitions from the theoretical concepts discussed in the previous chapter—such as the OAuth 2.0 roles defined in RFC 6749 [3] and the Casbin PERM metamodel [23]—to their concrete application. We will detail the integration between the key entities: the OpenC2 **Producer**, which acts as the OAuth 2.0 Client; the **Consumer**, which functions as the Resource Server and **Policy Enforcement Point (PEP)**; the **Authorization Server (AS)**, responsible for authentication and token issuance; and the **User Agent (UA)**, which facilitates the operator's interaction with the system.

The implementation primarily utilizes the Python programming language, and the entire framework is built as an extension of the `otupy` library [4]. The web-based components, including the Authorization Server and the User Agent, are built using the Flask web framework. The core OAuth 2.0 logic is handled by the `Authlib` library [26], while the `Casbin` library [23] is integrated into the Consumer to provide fine-grained authorization based on the policies discussed in the previous chapter.

The following sections will delve into the overall system architecture, the end-to-end communication flow, and the specific implementation details of each component.

6.2 System Architecture

The implemented system's architecture is designed to integrate the OAuth 2.0 authorization framework into the existing OpenC2 structure [2]. It is composed of four primary functional blocks: the **Producer**, the **Consumer**, the **Authorization Server (AS)**, and the **User Agent (UA)**. These components communicate over a **Transfer Layer**, which in our implementation supports both HTTP and MQTT. Figure 6.1 provides a high-level overview of this architecture, illustrating the interactions between each component.

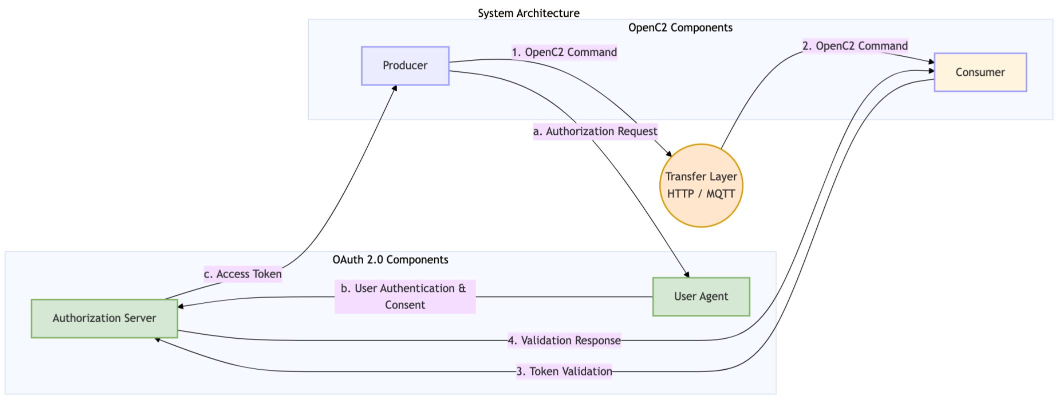


Figure 6.1: High-level architecture of the OAuth 2.0 integration in OpenC2.

6.2.1 Functional Blocks

- **Producer**: The Producer is an OpenC2 component responsible for creating and sending OpenC2 commands to a Consumer. In our security-enhanced architecture, it is modified to handle authentication flows, manage access tokens, and attach them to outgoing commands.

- **Consumer**: The Consumer is the recipient of OpenC2 commands. It acts as the guardian of the protected resources (i.e., the cyber defense functions it controls). Its primary security role is to receive commands, validate the attached access token, and enforce authorization policies before executing the requested `action` on a given `target`.

- **Authorization Server (AS)**: This is a new, centralized component dedicated to managing security. It is responsible for authenticating the identity of the human operator (the Resource Owner) and, upon successful authentication, issuing access tokens to the Producer (the Client).
- **User Agent (UA)**: The User Agent is a web-based intermediary that facilitates the authentication process. Since the Producer is often a command-line application or a background service without a direct user interface, the UA provides the necessary web interface for the operator to interact with the AS and grant the Producer permission to obtain an access token.
- **Transfer Layer**: This layer is responsible for the transport of OpenC2 messages between the Producer and Consumer. Our implementation supports both HTTP [17] and MQTT, with the security token embedded in the protocol headers or message properties.

6.2.2 Trust Boundaries and Security

The architecture defines clear trust boundaries, with the **Authorization Server** acting as the central root of trust for authentication. The choice of communication protocols is tailored to the specific interaction:

- **Authentication Flow**: The entire authentication process, involving the *Producer*, *User Agent*, and *Authorization Server*, is conducted exclusively over **HTTP**. This is a fundamental requirement, as the OAuth 2.0 framework relies on a standard, redirect-based web flow that is native to HTTP.
- **Command and Control**: For the subsequent command and control messaging between the *Producer* and *Consumer*, the architecture is flexible, supporting both **HTTP** and **MQTT** as transport protocols.

Among the available grant types, the architecture adopts the **Authorization Code Grant**, as recommended by the *OAuth 2.0 Security Best Current Practice* [22]. This flow ensures a strong separation between the *front-channel* (used for redirections and user authentication) and the *back-channel* (used for secure token exchanges), thereby preventing access token exposure to the browser or user agent.

This approach provides three key guarantees:

- **Authentication**: Producers are verified through token-based credentials issued by the Authorization Server.

- **Delegation**: Authorization is granted according to explicit user or system consent, limited by scope and token lifetime.

- **Trust Foundation**: The token acts as a verifiable proof of identity and permission, serving as the basis for fine-grained policy enforcement.

By adopting the Authorization Code Grant, the OpenC2 communication model gains a secure and interoperable mechanism for authenticating Producers and validating their privileges.

In a production environment, all these communication channels must be secured using **Transport Layer Security (TLS)**, i.e., HTTPS and MQTTS. This is a critical requirement to ensure the confidentiality and integrity of all exchanges, including redirects, authorization codes, access tokens, and OpenC2 commands, thereby preventing eavesdropping and man-in-the-middle attacks. The *Consumer* operates under the assumption that it can trust the *Authorization Server*, verifying the authenticity and validity of every access token through a secure introspection call.

For the scope of this thesis, however, the communication channels were implemented using plain **HTTP** and **MQTT** without TLS. This decision was made to simplify the development and testing setup, allowing the focus to remain on validating the core authentication and authorization logic provided by OAuth 2.0 and Casbin, rather than on the complexities of managing a Public Key Infrastructure (PKI). The migration to a secure setup is straightforward and discussed further in Section 6.8.

6.2.3 Mapping to OAuth 2.0 Roles

To align our implementation with the standard OAuth 2.0 framework [3], we mapped each architectural component to a specific OAuth 2.0 role. This mapping clarifies the responsibilities of each component within the authorization flow.

- **Producer → Client**: The Producer is the application that requests access to a protected resource on behalf of the user. It initiates the authorization flow and uses the access token to authenticate its command messages.

- **Consumer → Resource Server**: The Consumer hosts the protected resources (the cyber defense functions) and accepts access tokens from the Producer, which it must validate before granting access.

- **Authorization Server (AS) → Authorization Server**: This is a direct mapping. The AS is responsible for the entire authorization process, from authenticating the user to issuing tokens.

- **User Agent (UA) / Operator → Resource Owner**: The human operator, interacting via the UA, is the Resource Owner. They own the right to grant or deny access to the protected resources and delegate this right to the Producer by authenticating with the AS.

6.3 End-to-End Flow

The entire authentication and command-dispatch process is orchestrated through a sequence of interactions between the four main components. This flow is designed to be initiated automatically when the Producer attempts to send an OpenC2 command without a valid access token.

The process is based on the **OAuth 2.0 Authorization Code grant type** [3], which is considered the most secure flow for confidential clients. However, the standard Authorization Code flow is designed for traditional web applications, where a human user operating a web browser grants an external application (the Client) access to their resources (hosted on a Resource Server).

In our OpenC2 architecture, this model is adapted. The **Producer** (the Client) is a command-line application or a background service, not a web application. The operator (the Resource Owner) is interacting with a console, not a browser. This scenario presents a challenge, as the Authorization Code flow fundamentally relies on HTTP redirects, login forms, and user consent screens—all of which are browser-based interactions.

To bridge this gap, our architecture introduces the **User Agent (UA)** component. The UA acts as a "headless browser" or an automated agent for the console-based operator. It effectively "bypasses" the need for a manual browser session by programmatically handling the redirects and credential submissions required to complete the OAuth 2.0 flow, allowing a console application to securely obtain an access token.

The complete sequence of operations, including this modified flow, is illustrated in the sequence diagram in Figure 6.2. The following subsections provide a detailed, step-by-step explanation of this flow.

6.3.1 Detailed Step-by-Step Description

The process begins when a Producer needs to send a command but does not yet possess a valid access token.

1. **Initial Command and Unauthorized Response**: The Producer attempts to send an OpenC2 command to the Consumer's endpoint. Since the Producer does not yet have a valid access token, the transmission lacks the required

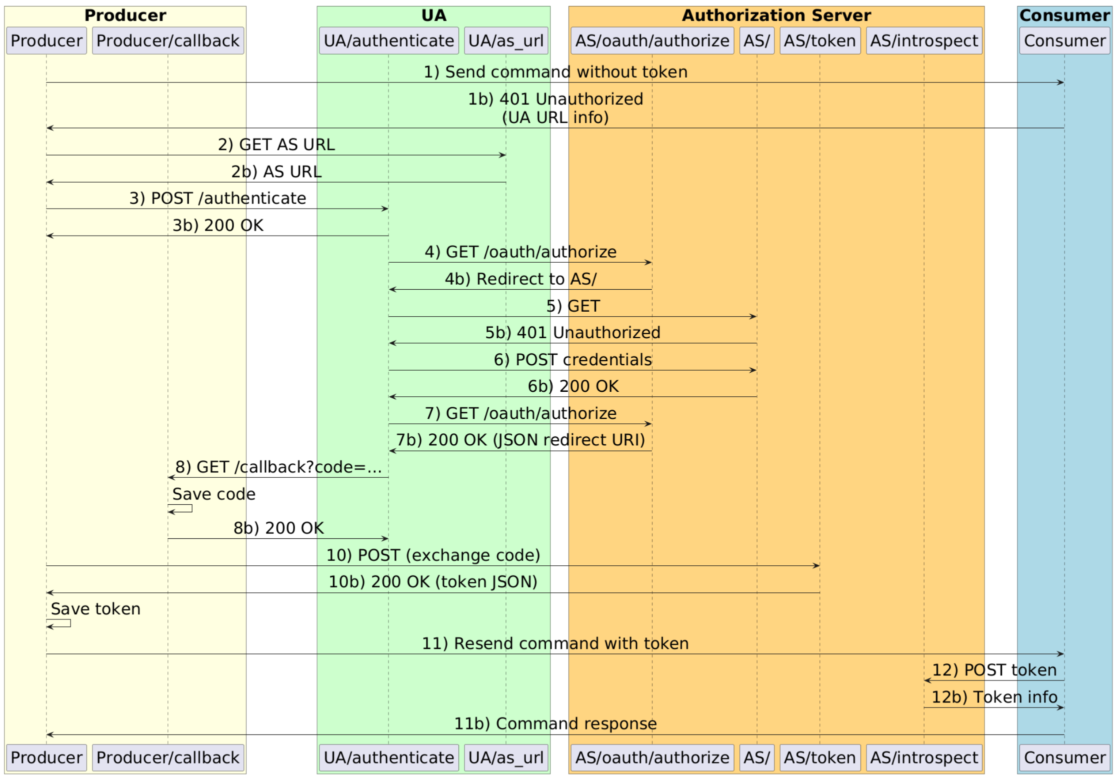


Figure 6.2: Sequence diagram of the complete authentication and command execution flow.

authentication credentials. The specific rejection mechanism depends on the transport protocol:

- When using **HTTP**, the absence of an `Authorization` header prompts the Consumer to respond with a `401 Unauthorized` status code.
- When using **MQTT**, the command is published without a token. The Consumer, failing to validate the message, constructs an OpenC2 response with a `401` status and publishes it to the `response_topic` specified by the Producer.

In both scenarios, the body of the response message crucially contains the URL of the User Agent (UA), enabling the Producer to initiate the authentication flow.

2. **Discovery of the Authorization Server**: The Producer's authentication module parses the `401` response to extract the UA's URL and makes a `GET` request to its `/as_url` endpoint. This allows the Producer to dynamically learn the location of the Authorization Server.

3. **Initiating Authentication with the User Agent**: The Producer sends a `POST` request to the UA's `/authenticate` endpoint to signal the start of the user-mediated authentication process.

4. **First Authorization Request**: The UA, on behalf of the Producer, constructs an authorization request and sends it to the AS's `/oauth/authorize` endpoint. This request includes the `client_id`, `redirect_uri`, `scope`, and a `state` parameter for CSRF protection. Since the operator is not yet authenticated, the AS redirects to its own login page.

5. **User Login Challenge**: The UA follows the redirect and presents the AS's login page to the operator, where the operator must prove their identity.

6. **Operator Submits Credentials**: The operator enters their username and password into the login form. The UA sends these credentials in a `POST` request to the AS's login endpoint. If successful, the AS establishes an authenticated session.

7. **Second Authorization Request and Consent**: The UA re-sends the original request to the `/oauth/authorize` endpoint. Now authenticated, the AS may present a consent screen. Upon the operator's approval, the AS generates a single-use authorization code.

8. **Callback to the Producer**: The AS redirects the UA to the Producer's pre-registered callback endpoint (`/callback`), including the generated `code` and the `state` parameter. The Producer saves the code and validates the `state`.

9. **Token Exchange**: The Producer makes a direct, server-to-server `POST` request to the AS's `/token` endpoint, authenticated with its client credentials and including the authorization `code` it just received. The AS validates the code and the client's identity.

10. **Token Issuance**: If validation is successful, the AS generates an **access token** and an optional **refresh token**, returning them in a JSON response. The Producer securely stores these tokens for future use.

11. **Authenticated Command and Introspection**: The Producer resends the command, this time including the access token (in the `Authorization` header for HTTP or in message properties for MQTT). The Consumer extracts the token and validates it by sending it to the AS's `/introspect` endpoint.

12. **Token Validation and Command Execution**: The AS's introspection endpoint verifies the token and returns an `"active": true` status if it is

valid. The Consumer then proceeds to the authorization phase (handled by Casbin) and, if authorized, executes the command.

6.3.2 HTTP Call Mapping

The table below summarizes the key HTTP interactions during the authentication flow.

Step	**Source**	**Destination**	**Description**
1	Producer	Consumer	Send command without token; receives 401.
2	Producer	UA	`GET /as_url` to discover the AS.
3	Producer	UA	`POST /authenticate` to start the flow.
4	UA	AS	`GET /oauth/authorize` (redirects to login).
5–6	UA / Operator	AS	Operator authenticates via login form.
7	UA	AS	`GET /oauth/authorize` again; gets auth code.
8	AS	Producer	Redirect to `/callback` with auth code.
9	Producer	AS	`POST /token` to exchange code for token.
11	Producer	Consumer	Send command with `Authorization` header.
11b	Consumer	AS	`POST /introspect` to validate the token.
12	Consumer	-	Execute command after successful validation.

Table 6.1: Mapping of HTTP calls to the roles involved in the end-to-end flow.

6.4 Authorization Server (AS) Implementation

The Authorization Server is the cornerstone of the OAuth 2.0 framework, responsible for authenticating the resource owner and issuing access tokens. While many robust, enterprise-grade Authorization Servers already exist (such as Keycloak, which, as discussed in the performance evaluation chapter, was later used for benchmarking), this thesis adopts a different approach for the initial design and validation phase.

This choice was primarily methodological. For the initial design, implementation, and functional validation of the framework presented in this chapter, adopting a full-scale Authorization Server would have introduced significant configuration and deployment complexity, potentially obscuring the core integration logic between the OpenC2 components.

Instead, a minimal yet standards-compliant AS was implemented using Flask and the `Authlib` library [26], in order to obtain a lightweight, transparent, and fully controllable environment. This setup was essential to support rapid prototyping, detailed inspection of the OAuth 2.0 flows, and effective debugging of the interactions among Producer, User Agent, and Consumer.

Once the *correctness* of the proposed architecture was validated using this custom AS, it was subsequently replaced with Keycloak to evaluate the *performance* and scalability of the solution against a realistic, production-grade reference.

The resulting lightweight AS is implemented as a standalone web application based on Flask, providing a flexible foundation that remains compliant with the relevant RFCs [3, 27].

6.4.1 Technology Stack

The choice of Flask was motivated by its simplicity and extensibility, making it ideal for creating a dedicated service with a clear set of responsibilities. `Authlib` was selected for its comprehensive implementation of the OAuth 2.0 specification, including support for the Authorization Code Grant, token introspection, and token revocation. Data persistence is managed using Flask-SQLAlchemy, which allows for easy integration with a relational database like SQLite for development.

6.4.2 Implemented Endpoints

The AS exposes a set of standard OAuth 2.0 endpoints, implemented as Flask routes:

- `/oauth/authorize`: Handles the initial authorization request. It authenticates the resource owner and obtains their consent before issuing an authorization code.
- `/token`: The endpoint where the client exchanges an authorization code for an access token. The request must be authenticated with the client's credentials.
- `/introspect`: As defined in RFC 7662 [27], this endpoint allows Resource Servers (OpenC2 Consumers) to validate an access token.
- `/`: This endpoint serves the user login page and handles the operator's credentials.

6.4.3 Database Schema

A persistent storage backend is crucial for managing state. The implementation uses a relational database with four main tables defined as SQLAlchemy models:

- **User**: Stores resource owner credentials (username and hashed password).
- **OAuth2Client**: Contains registration information for each Producer, including its `client_id`, `client_secret`, `redirect_uris`, and authorized scopes.

- **OAuth2AuthorizationCode**: Temporarily stores the authorization codes. Once used, a code is invalidated.
- **OAuth2Token**: Stores the issued access and refresh tokens, linked to a user and a client, including expiration and status (active or revoked).

6.4.4 Client Registration

Before a Producer can participate in an OAuth 2.0 flow, it must be registered as a client with the Authorization Server. This process involves storing the client's metadata in the database, primarily in the `OAuth2Client` table. While the OAuth 2.0 Dynamic Client Registration Protocol [28] defines a standard API for this, our implementation uses a simpler, manual registration method for populating the initial client data.

The necessary client metadata includes:

- `client_id` and `client_secret`: Credentials for the client to authenticate itself to the AS.
- `redirect_uris`: A list of allowed callback URLs to which the AS can send the authorization code.
- `grant_types`: The grant types the client is permitted to use (e.g., `authorization_code`).
- `response_types`: The response types the client expects (e.g., `code`).
- `scope`: The scopes the client is allowed to request.

The following code snippet demonstrates how a new client can be programmatically added to the database. This script is typically run once to set up the trusted clients of the Authorization Server.

```python
# Script to add a new OAuth2 client to the database
# Based on: examples/oauth2_examples/AS/website/models.py
from website.app import app, db
from website.models import OAuth2Client

def create_client():
    with app.app_context():
        # Example client metadata
        client_metadata = {
            'client_name': 'OpenC2 Producer',
            'client_uri': 'http://localhost:5002/',
            'grant_types': ['authorization_code', 'refresh_token'],
            'redirect_uris': ['http://localhost:5002/callback'],
            'response_types': ['code'],
            'scope': 'openc2',
            'token_endpoint_auth_method': 'client_secret_basic'
        }

        # The client_id and client_secret are typically generated
        # and stored securely.
        client = OAuth2Client(
            client_id='my-producer-client-id',
            client_secret='my-super-secret-key',
            **client_metadata
        )

        # Set client metadata for Authlib
        client.set_client_metadata(client_metadata)

        db.session.add(client)
        db.session.commit()
        print("Client registered successfully.")

if __name__ == '__main__':
    create_client()
```

6.4.5 Security Mechanisms

The implementation incorporates several security mechanisms, adapted for the development environment.

- **TLS**: Endpoints are designed for HTTPS. However, for simplicity during development, TLS was not activated. To run the server over HTTP, the `Authlib` library's default HTTPS enforcement was disabled by setting the environment variable `AUTHLIB_INSECURE_TRANSPORT=1`.
- **State Parameter**: The use of a randomized `state` parameter is enforced during the authorization flow to prevent Cross-Site Request Forgery (CSRF) attacks.

It is important to note that Proof Key for Code Exchange (PKCE) [21], a recommended security measure, has not been implemented in the current version of the code.

6.4.6 Code Snippet: Grant Registration

The following snippet from `website/oauth2.py` illustrates how `Authlib` is used to configure the AS and register the supported grant types.

```python
# File: examples/oauth2_examples/AS/website/oauth2.py

from authlib.integrations.flask_oauth2 import AuthorizationServer
from .models import db, User, OAuth2Client, OAuth2Token
from .oauth2_grants import (
    AuthorizationCodeGrant,
    RefreshTokenGrant,
)

def config_oauth(app):
    query_client = OAuth2Client.get_by_client_id
    save_token = OAuth2Token.save

    server = AuthorizationServer(
        app,
        query_client=query_client,
        save_token=save_token
    )

    # Register the supported grant types for the token endpoint
    server.register_grant(AuthorizationCodeGrant)
    server.register_grant(RefreshTokenGrant)
```

6.5 User Agent (UA) Implementation

The User Agent (UA) is a critical component that bridges the gap between the typically non-interactive Producer and the authentication flow of the Authorization Server. Since the Producer is often a command-line tool or a background service, it lacks the web browser interface required for an operator to enter credentials and grant consent. The UA is implemented as a lightweight Flask web application that acts as a temporary, headless browser on behalf of the Producer, orchestrating the necessary HTTP requests and redirects to complete the authentication process.

6.5.1 Role and Responsibilities

The primary role of the User Agent is to act as an intermediary, managing the complex sequence of interactions required by the Authorization Code grant. Its key responsibilities include:

- **Service Discovery**: Providing the Producer with the necessary endpoint information for the Authorization Server.
- **Authentication Orchestration**: Handling the redirect-based authentication

flow, which involves receiving the operator's credentials, submitting them to the AS, and managing the session cookies to maintain an authenticated state.

- **Consent and Redirection**: Forwarding the authorization request to the AS and, upon successful authentication and consent, relaying the final authorization code back to the Producer's callback endpoint.

6.5.2 Main Endpoints

The User Agent exposes a minimal set of endpoints to fulfill its role. These are defined in the `UA.py` file within the 'otupy' source code.

- `/as_url`: This is a simple discovery endpoint. When the Producer first receives a `401 Unauthorized` response, it contacts this endpoint via a `GET` request to dynamically obtain the base URL of the Authorization Server. This decouples the Producer from the AS, avoiding hardcoded configurations.

- `/authenticate`: This is the main functional endpoint. It is triggered by a `POST` request from the Producer to initiate the authentication flow. Upon receiving this request, the UA starts a new authentication session, constructs the initial authorization URL with the required parameters (`client_id`, `redirect_uri`, `scope`, `state`), and begins the series of HTTP requests to the AS to log the user in and obtain the authorization code.

6.5.3 Session Management and Authentication Flow

The UA's implementation leverages the `requests` library to maintain a persistent session throughout the authentication flow. This is crucial for handling the session cookies set by the Authorization Server after a successful login.

The process managed by the UA is as follows:

1. A `requests.Session` object is created to ensure that cookies are persisted across subsequent requests to the AS.

2. The UA first makes a request to the AS's `/oauth/authorize` endpoint. Since the session is not yet authenticated, the AS responds with a redirect to its login page.

3. The UA follows the redirect and then programmatically submits the operator's credentials (which could be pre-configured or passed to the UA) via a `POST` request to the AS's login endpoint.

4. Upon successful login, the AS sets a session cookie. The UA's session object automatically stores this cookie.

5. The UA then re-sends the original request to the `/oauth/authorize` endpoint. This time, because the request includes the valid session cookie, the AS recognizes the operator as authenticated and proceeds to issue the authorization code, redirecting the UA to the Producer's callback URL.

6. Finally, the UA makes a request to this callback URL, delivering the authorization code to the Producer and thus completing its role in the flow.

6.5.4 Code Snippet: User Agent Endpoints

The following code snippet, extracted from the `src/otupy/apps/oauth2/UA/UA.py` file, shows the implementation of the main Flask routes of the User Agent. The logic for handling the authentication flow is encapsulated within the `AuthAgent` class, which is instantiated and used within the `/authenticate` route.

```python
# File: src/otupy/apps/oauth2/UA/UA.py

from flask import Flask, request, jsonify
from AuthAgent import AuthAgent
import os

app = Flask(__name__)

# Load configuration from environment variables
AS_URL = os.environ.get("AS_URL")
PRODUCER_CALLBACK_URL = os.environ.get("PRODUCER_CALLBACK_URL")
CLIENT_ID = os.environ.get("CLIENT_ID")
CLIENT_SECRET = os.environ.get("CLIENT_SECRET")
USERNAME = os.environ.get("USERNAME")
PASSWORD = os.environ.get("PASSWORD")

auth_agent = AuthAgent(
    as_url=AS_URL,
    redirect_uri=PRODUCER_CALLBACK_URL,
    client_id=CLIENT_ID,
    client_secret=CLIENT_SECRET,
    username=USERNAME,
    password=PASSWORD
)

@app.route('/as_url', methods=['GET'])
def get_as_url():
    """
    Endpoint for the Producer to discover the Authorization Server URL.
    """
    return jsonify({"as_url": AS_URL})

@app.route('/authenticate', methods=['POST'])
def authenticate():
    """
    Initiates the authentication process on behalf of the Producer.
    """
    try:
        auth_agent.authenticate()
        return "Authentication process started.", 200
    except Exception as e:
        return f"An error occurred: {e}", 500

if __name__ == '__main__':
    app.run(port=5001, debug=True)
```

6.6 Producer Implementation

The OpenC2 Producer, acting as the OAuth 2.0 Client, is the initiator of both the authentication flow and the command-and-control sequence. Its implementation is a critical piece of the architecture, as it must seamlessly integrate the security mechanisms without disrupting the core functionality of creating and dispatching OpenC2 commands.

6.6.1 Architectural Design and Core Components

The Producer is designed as a command-line application that leverages the existing `otupy` library's `Producer` class [4]. The key to integrating the OAuth 2.0 flow is the introduction of a modular **Authenticator**. This design choice decouples the authentication logic from the Producer's primary responsibility of sending commands.

The authentication mechanism is encapsulated within the `OAuth2Authenticator` class, which inherits from a generic `Authenticator` abstract base class defined in `src/otupy/auth/Authenticator.py`. This abstract class defines a simple interface, ensuring that different authentication methods can be implemented and swapped without altering the Producer's core logic.

```
# File: src/otupy/auth/Authenticator.py
class Authenticator:
    def authenticate(self, auth_endpoint): pass
    def is_authenticated(self): pass
```

The `OAuth2Authenticator` class implements this interface, handling all the steps of the Authorization Code Grant flow described in Section 6.3. It manages the discovery of the AS, the interaction with the UA, the handling of the callback, the exchange of the authorization code for an access token, and the secure storage of that token.

6.6.2 Producer Initialization and Configuration

The main application logic is contained within `examples/oauth2_examples /producer_oauth2.py`. During initialization, the Producer is configured with the necessary components: an encoder (e.g., `JSONEncoder`), a transfer mechanism (e.g., `HTTPTransfer`), and, most importantly, an instance of the `OAuth2Authenticator`.

The `OAuth2Authenticator` is instantiated with the client's credentials and the redirect URI for the callback endpoint. This information is used to identify the Producer to the Authorization Server and to correctly route the authorization code back to the Producer. The following code snippet from `producer_oauth2.py` illustrates this setup.

```python
# File: examples/oauth2_examples/producer_oauth2.py

# ... imports ...

def main():
    """Create an OAuth2 Producer and send commands"""

    # Keycloak configuration for OAuth2
    oauth2_config = {
        'client_id': 'Producer',
        'client_secret': 'RYuumxqGJwXTZP52Of5ImUcWEA4TnTUa',
        'redirect_uri': 'http://127.0.0.1:8000/callback',
        'callback_port': 8000
    }
    oauth2authenticator = OAuth2Authenticator(**oauth2_config)

    producer = Producer(
        producer="producer.example.net",
        encoder=JSONEncoder(),
        transfer=HTTPTransfer("127.0.0.1", 9000),
        authenticator=oauth2authenticator
    )

    # ... command creation ...

    producer.sendcmd(cmd3)
```

6.6.3 Triggering the Authentication Flow

The authentication process is transparently triggered by the `sendcmd()` method of the `Producer` class. When `sendcmd()` is called, it first checks if the provided authenticator has a valid access token.

- **If a valid token is not available**, the `Producer` calls the `authenticate()` method on its `authenticator` instance. This initiates the entire end-to-end flow described in Section 6.3. The `OAuth2Authenticator` will then guide the operator through the login and consent process via the User Agent, ultimately obtaining an access token from the Authorization Server.
- **If a valid token is already present**, the `Producer` proceeds directly to dispatching the command, attaching the token to the request. The token is included in the `Authorization` header for HTTP requests, following the Bearer Token scheme [19].

This lazy-initialization approach to authentication ensures that the user is only prompted to log in when necessary, and that subsequent commands can be sent using the stored token until it expires. The code snippet below, from the main execution block of `producer_oauth2.py`, shows the creation of an OpenC2 command and the call to `sendcmd()` that initiates this process.

```
# File: examples/oauth2_examples/producer_oauth2.py

# ... (inside main function)

        actuator_profile = slpf.Specifiers({
            'hostname': 'firewall',
            'named_group': 'firewalls',
            'asset_id': 'iptables'
        })

        args = slpf.Args({'response_requested': oc2.ResponseType.complete})

        # Example command: allow traffic from a specific subnet
        cmd3 = oc2.Command(
            oc2.Actions.allow,
            oc2.IPv4Net('130.0.16.0'),
            args,
            actuator=actuator_profile
        )

        producer.sendcmd(cmd3)
```

```
# ... (exception handling) ...

if __name__ == "__main__":
    logger.info("Starting OpenC2 Producer with OAuth2...")
    logger.info("Sending Command...")
    main()
```

The modular design, centered around the `Authenticator` interface, provides a clean and extensible way to secure the OpenC2 Producer, making the authentication process a transparent and integrated part of the command execution workflow.

6.7 Consumer Implementation

The OpenC2 Consumer, functioning as the OAuth 2.0 Resource Server, is the component responsible for receiving and processing OpenC2 commands. Its primary security role is to act as a gatekeeper for the protected resources it controls, enforcing both **authentication** (via OAuth 2.0) and **authorization** (via Casbin) before executing any action.

6.7.1 Architectural Design and Core Components

In contrast to the Producer, the Consumer is a passive participant in the authentication flow. It does not initiate authentication but is responsible for validating the credentials, the access token, provided with each incoming command.

To centralize this security logic, the implementation extends the generic `Authorizer` class into a more specific `OAuth2Authorizer` class. This design maintains a separation of concerns, keeping the security validation logic distinct from the Consumer's core task of command processing and actuator management. The `OAuth2Authorizer` is responsible for two critical security checks:

- **Token Introspection (Authentication)**: It validates the received access token by communicating with the Authorization Server's introspection endpoint [28]. This step verifies that the token is active, has not expired, and was issued by a trusted AS.

- **Authorization Enforcement**: After successful authentication, it enforces access control policies using Casbin. It evaluates whether the identity associated with the token is permitted to perform the requested OpenC2 action on the specified `target`.

6.7.2 Consumer Initialization and Configuration

A secure Consumer is instantiated by providing it with an instance of the OAuth2Authorizer during its setup. The authorizer itself is configured with the necessary security parameters, including the URL of the Authorization Server (for token introspection), the URL of the User Agent (to be returned in case of failed authentication), and the paths to the Casbin model and policy files. The following code snippet illustrates how a secure Consumer would be configured and initialized. The `HTTPTransfer` layer is configured to pass incoming requests to the `OAuth2Authorizer` for validation before they are processed by the Consumer's main logic.

```python
# File: examples/oauth2_examples/consumer_oauth2.py (logic adapted)

from otupy.core.consumer import Consumer
from otupy.transfers.http import HTTPTransfer

from otupy.oauth2.OAuth2Authorizer import OAuth2Authorizer
# ... other imports

def main():
    """Create and run a secure OpenC2 Consumer."""

    # Unified configuration for the OAuth2Authorizer
    auth_config = {
        'as_url': 'http://127.0.0.1:5000',
        'ua_url': 'http://127.0.0.1:5001',
        'model': 'src/otupy/apps/oauth2/UA/model.conf',
        'policy': 'src/otupy/apps/oauth2/UA/policy.csv'
    }

    # Initialize the single Authorizer component
    authorizer = OAuth2Authorizer(**auth_config)


    consumer = Consumer(
        actuators={'slpf': MyFirewallActuator()},
        authorizer=authorizer
    )

    # Il transfer layer (HTTP) userà l'authorizer per
    # validare ogni richiesta in arrivo.
    http_server = HTTPTransfer("127.0.0.1", 9000)
    http_server.start(consumer)
```

```
if __name__ == "__main__":
    logger.info("Starting OpenC2 Consumer with OAuth2...")
    main()
```

6.7.3 Casbin Authorization Implementation

The integration of Casbin within the otupy framework is designed to centralize authorization logic, making it both maintainable and extensible. The authorization process is managed by the Authorizer class, located in src/otupy/auth/Authorizer.py [29]. For this implementation, a standard Role-Based Access Control (RBAC) model was selected. This paradigm offers an ideal balance between simplicity and expressive power, naturally fitting the operational requirement of assigning distinct permissions to different classes of OpenC2 actors, such as administrators and standard users.

Model and Policy Configuration

The authorization model for the OpenC2 Consumer is defined in src/otupy/apps/oauth2/UA/model.conf. It implements a standard RBAC model with role inheritance, as shown below:

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[role_definition]
g = _, _

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
```

This model establishes that a request is authorized if the requesting subject (r.sub) is a member of a role (p.sub) that has permission to perform the requested action (r.act) on the specified object (r.obj).

The corresponding policies are defined in src/otupy/apps/oauth2/UA/policy.csv. This file contains both policy rules (p) and role assignments (g):

```
p, admin, openc2, allow
```

```
p, user, openc2_command, allow

g, alice, admin
g, bob, user
```

In this configuration:

- The ‘admin‘ role is granted permission to perform any action (‘allow‘) on the ‘openc2‘ object.
- The ‘user‘ role has a more restricted permission, limited to the ‘openc2_command‘ object.
- User ‘alice‘ is assigned the ‘admin‘ role, while user ‘bob‘ is assigned the ‘user‘ role.

The Base Authorization Workflow

The `Authorizer` base class encapsulates the core logic in its `enforce` method, which receives the subject, object, and action and returns a boolean decision. This clean separation allows the authorization rules to be updated by simply modifying the `policy.csv` file.

The following code snippet from `src/otupy/auth/Authorizer.py` illustrates this base implementation:

```python
import casbin

class Authorizer:
    def __init__(self, model_path, policy_path):
        self.enforcer = casbin.Enforcer(model_path, policy_path)

    def enforce(self, sub, obj, act):
        """
        Checks if a subject has permission to perform an action
        on an object.
        """
        return self.enforcer.enforce(sub, obj, act)
```

This clean separation allows the authorization rules to be updated by simply modifying the policy.csv file, without requiring any changes to the Python source code. This is particularly advantageous in a dynamic security environment where permissions may need to be altered frequently in response to evolving threats or operational requirements.

6.7.4 The Two-Step Validation Process

The enforcement of security policies is triggered automatically by the transfer layer for every incoming command.

- **If a request contains a valid access token**, the `OAuth2Authorizer` performs the two-step validation. It first sends the token to the AS for introspection. If the token is active, it then uses the identity information (e.g., the `sub` claim from a JWT or the introspection response) to perform a Casbin `enforce()` check. Only if both checks pass is the command forwarded to the appropriate actuator for execution.

- **If a request is missing a token or the token is invalid** (expired, revoked, or unknown), the introspection fails. The authorizer then instructs the transfer layer to reject the command with a `401 Unauthorized` status. Crucially, the body of this response contains the URL of the User Agent, providing the Producer with the entry point to initiate the authentication flow as described in Section 6.3.

The code snippet below, from `src/otupy/oauth2/OAuth2Authorizer.py`, shows the implementation of the core validation logic, combining token introspection with Casbin policy enforcement.

```python
# File: src/otupy/oauth2/OAuth2Authorizer.py (logic adapted)
class OAuth2Authorizer(CasbinAuthorizer):
    def __init__(self, as_url, ua_url, model, policy):
        self.as_url = as_url
        self.ua_url = ua_url
        self.enforcer= casbin.Enforcer(model,policy)

    def is_authorized(self, access_token: str, command: Command) -> bool:
        """
        Verify if the token is valid and the command is authorized.
        """
        if not self._is_token_valid(access_token):
            return False
        # Placeholder for extracting subject (user) from a JWT token
        sub = "user_identity_from_token"

        # Extract action and target for Casbin enforcement
        act = command.action.value
        obj = str(command.target)

        return self.enforcer.enforce(sub, obj, act)
```

```
def _is_token_valid(self, access_token: str) -> bool:
    """
    Introspects the token with the Authorization Server.
    """
    introspection_url = f"{self.as_url}/introspect"
    try:
        response = requests.post(introspection_url,
                                 data={'token': access_token})
        if response.status_code != 200:
            return False

        return response.json().get('active', False)
    except requests.exceptions.RequestException:
        return False
```

This modular approach ensures that the Consumer robustly enforces security policies, protecting its resources while providing the necessary feedback to unauthorized clients to enable proper authentication.

6.8 Security of Transfer Layers and Tokens

The security of the authentication framework does not solely rely on the OAuth 2.0 protocol but extends to the practical implementation details of the communication channels and the tokens used. While TLS was disabled during development for simplicity, its enforcement is non-negotiable in a production environment. This section expands on the security considerations for the transfer layer and discusses the critical aspects of token management, including token format choices and their lifecycle.

6.8.1 Channel Security

As established, all communication channels must be secured using **Transport Layer Security (TLS)** to ensure confidentiality and integrity. This applies to:

- The entire authentication flow between the Producer, User Agent, and Authorization Server (i.e., HTTPS).
- The command and control messaging between the Producer and Consumer (i.e., HTTPS or MQTTS).
- The token introspection calls from the Consumer to the Authorization Server (i.e., HTTPS).

Without TLS, sensitive data such as authorization codes, access tokens, and the content of OpenC2 commands would be transmitted in cleartext, making them vulnerable to eavesdropping and man-in-the-middle attacks.

For HTTP-based communication, migrating to HTTPS is achieved by configuring a valid X.509 certificate on the server. In a Flask application, this can be done as follows:

```
app.run(ssl_context=('cert.pem', 'key.pem'))
```

Similarly, for MQTT, secure transport (MQTTS) is enabled by configuring a TLS-capable broker (e.g., `mosquitto`) and setting up the necessary certificate files in the client:

```
client.tls_set(ca_certs="ca.crt", certfile="client.crt", keyfile="client.key")
client.connect("broker.example.com", 8883)
```

6.8.2 Token Format: Opaque vs. JWS

An access token is the credential used by the Producer to authenticate its requests to the Consumer. The OAuth 2.0 specification does not mandate a specific token format, leaving the decision to the implementation. The two primary approaches are opaque tokens and self-contained tokens, such as JSON Web Signature (JWS).

- **Opaque Tokens**: These are random strings that do not contain any user or permission information. An opaque token acts as a reference to the authorization data stored securely on the Authorization Server. To validate an opaque token, the Consumer must send it back to the AS via a dedicated introspection endpoint. This was the approach taken in our implementation, as it is simple and highly secure. Since the token itself contains no data, there is no risk of information leakage if a token is intercepted.

- **JSON Web Signature (JWS) Tokens**: JWT is a standard for creating self-contained, digitally signed tokens. A JWS token is typically a JSON Web Token (JWT), which consists of three parts: a header, a payload, and a signature. The payload contains claims, such as the user's identity (`sub`), the token's expiration time (`exp`), and the authorized scopes. The token is signed by the AS with a private key. The Consumer, possessing the corresponding public key, can verify the token's authenticity and integrity locally without contacting the AS. This avoids the overhead of an introspection call for every request, improving performance. However, it also means the token's claims are readable (though not modifiable) by anyone who intercepts it.

The choice between these formats involves a trade-off. Opaque tokens offer better security against information leakage but require a network call for every validation. JWS tokens are more performant for validation but require careful management of cryptographic keys and expose claims data.

6.8.3 Token Lifecycle Management

Regardless of the format, access tokens must be managed throughout their lifecycle to maintain security. This includes timely revocation and periodic rotation of the cryptographic keys used to sign them.

- **Token Revocation**: An access token should be invalidated before its natural expiration if it is compromised or if the user's session is terminated. For **opaque tokens**, revocation is straightforward: the AS simply marks the token as invalid in its database. The next time the Consumer attempts to introspect the token, the AS will report it as inactive. For **JWS tokens**, revocation is more complex because they are validated offline. To revoke a JWS, the Consumer must be able to check a revocation list (CRL) or use an online validation service, which reintroduces the network dependency that JWS aims to avoid.
- **Key Rotation**: When using JWS tokens, the keys used by the Authorization Server to sign the tokens must be periodically rotated. If a signing key is compromised, an attacker could forge valid tokens. A key rotation strategy ensures that even if a key is leaked, its useful lifetime is limited. The AS should expose a JSON Web Key Set (JWKS) endpoint where Consumers can dynamically fetch the current set of public keys for validating JWS signatures.

In our implementation, the use of opaque tokens simplifies lifecycle management, as both validation and revocation are handled centrally by the Authorization Server's database. This design prioritizes security and control over the performance benefits of stateless JWS tokens.

Chapter 7

Testing and Performance Evaluation

7.1 Introduction

After implementing the authentication and authorization framework, it is essential to validate its correctness and evaluate its performance. This chapter presents the testing methodology and the results obtained from two main sets of experiments. The primary goals are twofold:

1. **Functional Testing:** To verify that the security mechanisms work as intended, ensuring that access control policies are correctly enforced by the Consumer. This involves testing that authorized commands are executed while unauthorized ones are rejected.

2. **Performance Analysis:** To measure the overhead introduced by the OAuth 2.0 authentication and Casbin authorization layers. By comparing the system's performance with and without these security measures, we can quantify the latency they add to the command processing pipeline.

All the tests described in this chapter were conducted using the scripts and configurations available in the project's `validation` directory. The test environment for both functional and performance validation was hosted entirely on a single local machine to minimize latency and ensure a controlled, repeatable setup.

The OpenC2 Producer, the OpenC2 Consumer, and the User Agent (UA) were each bound to a separate local port, ensuring isolation between components without relying on external networks. The Authorization Server (AS) was implemented using a Keycloak instance running in a Docker container on the same host. This consistent setup allowed both functional validation and performance analysis to be executed under identical architectural conditions, providing a reliable and reproducible baseline for assessing correctness and measuring overhead.

7.2 Functional Testing: Access Control Validation

The first phase of testing focused on validating the correctness of the access control logic implemented in the Consumer. The objective was to ensure that the combined use of OAuth 2.0 token introspection and Casbin policy enforcement correctly filters incoming OpenC2 commands.

7.2.1 Test Setup and Methodology

The test campaign was orchestrated by the Python script `test_access_control.py`, which acts as an OpenC2 Producer. The script systematically sends predefined commands to the secured Consumer using the HTTP-based transport and the same OAuth 2.0 authentication mechanisms employed by the actual Producer.

The OpenC2 commands used for validation are loaded from the `validation/` `/otupy/oauth2/openc2-commands` directory. This directory contains **76** distinct command definitions, each representing a specific authorization scenario. Before executing the tests, the script obtains a valid access token for the user "admin" through the full OAuth 2.0 flow against the local Authorization Server. It then iterates over the entire command set, generating a new authorization check for each command against the secured Consumer. For each request, the script records the HTTP status code and relevant log information.

The Consumer is configured with the `OAuth2Authorizer`, which relies on the Casbin model defined in `model.conf` and the policies specified in `policy.csv` to make authorization decisions. An excerpt of the policy is reported below:

```
p, user, ipv4_net, allow
p, user, file, update
p, user, features:*, query
p, user, ipv6_net, deny
```

Listing 1: Example policies from `policy.csv`

The command set is intentionally heterogeneous and is designed to exercise both permitted and forbidden cases. In particular, it includes:

- commands targeting resources that are explicitly allowed (e.g. `ipv4_net`);
- commands targeting resources that are explicitly denied (e.g. `ipv6_net`);

- commands involving the `features` target, used to verify fine-grained permissions such as allowed `query` operations;
- variants where only specific fields (such as the target type or the requested operation) are modified, to confirm that even small deviations from the allowed action–target combinations are correctly rejected.

This setup ensures systematic coverage of the relevant policy rules and validates the interaction between token introspection, policy evaluation, and command execution under realistic usage conditions.

7.2.2 Results and Discussion

The execution of `test_access_control.py` confirmed that the access control mechanism behaves as expected across the entire test set. The logs collected in `controller.log` show that the Consumer's responses are consistent with the configured policies.
For each command sent, the system operated as follows:

- Commands whose action–target pair matched an applicable "allow" rule in `policy.csv` for the authenticated "user" were successfully authorized, and the Consumer returned a status of 200 OK.
- Commands for which no matching "allow" rule existed, or that were explicitly disallowed by policy, were correctly rejected by the Casbin enforcer, resulting in a 403 Forbidden status.

These results demonstrate that the `OAuth2Authorizer` correctly introspects the access token to identify the caller and that Casbin reliably enforces the defined authorization rules over OpenC2 commands. The functional correctness of the core security logic is therefore validated on a substantial and systematically constructed set of test cases.

7.3 Performance Analysis

While security is paramount, it is also important to understand the performance impact of the implemented framework. In particular, this analysis aims to quantify the overhead introduced by the OAuth 2.0 authentication and Casbin-based authorization layers when processing OpenC2 commands.

7.3.1 Test Methodology

To isolate the cost of the security mechanisms, two comparative tests were executed under identical conditions, using the same local environment described in Section 7.1.

In both scenarios, an OpenC2 Producer and an OpenC2 Consumer were instantiated on the same host and communicated via HTTP.
The performance evaluation is based on the same set of OpenC2 command definitions used in the functional testing phase. This ensures that any difference in execution time between the two scenarios can be attributed to the presence or absence of the security components, rather than to variations in the workload.
The scripts in the `validation/otupy/oauth2/test_performance/` directory implement the following two configurations:

1. **Baseline (No Authentication):** The `controller_no_auth.py` script sends the selected OpenC2 commands to a Consumer instance with no authentication or authorization enabled. This scenario measures the raw processing performance of the OpenC2 stack.

2. **Secure (OAuth 2.0 + Casbin):** The `controller_aouth.py` script sends the *same* set of commands to a Consumer configured with the `OAuth2Authorizer` and Casbin policies. For each incoming command, the Consumer validates the access token via an OAuth 2.0 token introspection request to the local Keycloak Authorization Server and enforces the authorization rules before executing the actuator logic.

In both cases, the Producer issues the same OpenC2 commands in the same order, ensuring a one-to-one comparability of the measured timings. For each request, the following metrics were collected:

- the end-to-end round-trip time (RTT) as observed by the Producer;

- the internal processing time on the Consumer side, decomposed into decoding, processing, and encoding stages, as recorded in the server logs.

The resulting log files were processed using AWK scripts to compute the mean, minimum, and maximum values for each metric.

7.3.2 Results and Discussion

The performance data, summarized in Table 7.1 and Table 7.2, highlights the latency overhead introduced by the security layers. All measurements are expressed in milliseconds (ms).

Table 7.1: Producer-side Performance Comparison (RTT). Data extracted from `controller-no-auth.txt` and `controller-auth.txt`.

Producer Metric (RTT)	**Baseline (ms)**	**Secure (ms)**	**Overhead (ms)**
Mean Tot. Transaction	4.88	19.35	14.47
Mean Send-to-Receive	3.67	17.40	13.73

Table 7.2: Consumer-side Processing Time Comparison. Data extracted from `server.txt` and `server-auth.txt`.

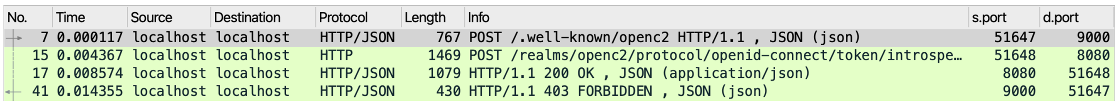
Consumer Metric	**Baseline (ms)**	**Secure (ms)**	**Overhead (ms)**
Mean Tot Processing	1.01	13.20	12.19
Mean Decoding	0.43	0.71	0.28
Mean Processing	0.37	11.86	11.49
Mean Encoding	0.21	0.63	0.42

From the Producer's perspective (Table 7.1), the mean round-trip time for a complete transaction increases from **4.88 ms** in the baseline scenario to **19.35 ms** in the secure scenario. This corresponds to an additional latency of 14.47 ms per command, i.e. a relative increase of approximately four times.
The Consumer-side measurements (Table 7.2) provide a more fine-grained view. The mean total processing time rises from **1.01 ms** to **13.20 ms**, introducing an overhead of 12.19 ms and resulting in a significant increase in the server's processing effort. The dominant contribution is captured by the "Mean Processing" metric: in the baseline case, the actuator-specific logic alone requires 0.37 ms on average, whereas in the secure configuration the same logical step is preceded by token introspection and Casbin policy evaluation, for a combined mean of 11.86 ms. The resulting 11.49 ms increase in this stage accounts for the majority of the server-side overhead and directly explains the RTT growth observed by the Producer.
This overhead is largely attributable to the per-request interaction with the Authorization Server (Keycloak), which in this setup runs as a Docker container on the same host. Despite the local deployment, the additional HTTP introspection call and the internal processing within Keycloak introduce measurable latency.
In summary, the comparison between the two scenarios, identical commands with and without security, shows that the adoption of OAuth 2.0 introspection and Casbin authorization introduces a consistent overhead of approximately 12–15 ms per transaction in this environment.
This represents an expected trade-off for strong, per-request validation. In high-throughput deployments, this cost could be reduced by introducing token caching

on the Consumer side, thereby limiting the number of introspection calls at the expense of slightly relaxed real-time revocation guarantees.

7.3.3 Attribution of Overhead via Network Traces

To clarify how the overhead manifests on the *end-to-end* path, we analyzed a single representative transaction in the secure configuration while capturing traffic on the loopback interface. **This measurement targets the full Producer→Consumer→Producer round trip** (i.e., the Producer's *send-to-receive* time) in steady state: the Producer already holds a valid access token, so the overhead comes from the Consumer's per-request token *introspection* against Keycloak plus local policy evaluation.



No.	Time	Source	Destination	Protocol	Length	Info	s.port	d.port
7	0.000117	localhost	localhost	HTTP/JSON	767	POST /.well-known/openc2 HTTP/1.1 , JSON (json)	51647	9000
15	0.004367	localhost	localhost	HTTP	1469	POST /realms/openc2/protocol/openid-connect/token/introspe…	51648	8080
17	0.008574	localhost	localhost	HTTP/JSON	1079	HTTP/1.1 200 OK , JSON (application/json)	8080	51648
41	0.014355	localhost	localhost	HTTP/JSON	430	HTTP/1.1 403 FORBIDDEN , JSON (json)	9000	51647

Figure 7.1: Wireshark capture of a secure OpenC2 transaction (Producer–Consumer–Keycloak–Consumer–Producer).

Sequence (steady state, token already present): (1) Producer→Consumer `POST /.well-known/openc2`; (2) Consumer→Keycloak `POST /realms/openc2/protocol/openid-connect/token/introspection`; (3) Keycloak→Consumer `HTTP/1.1 200 OK`; (4) Consumer→Producer `HTTP/1.1 403 FORBIDDEN`.
The capture reports the following timestamps (seconds, relative to the trace start):

- **(1)** `POST /.well-known/openc2` at $t_0 = 0.000117$;
- **(2)** `POST /realms/openc2/protocol/openid-connect/token/introspection` at $t_1 = 0.004367$;
- **(3)** `HTTP/1.1 200 OK` at $t_2 = 0.008574$;
- **(4)** `HTTP/1.1 403 FORBIDDEN` at $t_3 = 0.014355$.

From these timestamps we obtain the following breakdown (steady-state per-request behavior):

Pre-introspection local handling	$t_1 - t_0 \approx \mathbf{4.25\ ms}$
Keycloak introspection RTT	$t_2 - t_1 \approx \mathbf{4.21\ ms}$
Post-introspection local handling	$t_3 - t_2 \approx \mathbf{5.78\ ms}$
Total RTT (send → receive)	$t_3 - t_0 \approx \mathbf{14.24\ ms}$

Table 7.3: Packet-level breakdown for one secure RTT (Producer→Consumer→Producer).

Component	**Lat. (ms)**
Pre-introspection handling (parse, token extract, build call)	4.25
Introspection RTT (Consumer↔Keycloak)	4.21
Post-introspection handling (validate, Casbin, encode)	5.78
Total RTT (t_3–t_0)	**14.24**

Consistency with Producer *Send-to-Receive.* The packet-level timing above ($t_3 - t_0 \approx 14.24$ ms) is anchored at the Consumer (arrival→reply) and therefore excludes the Producer↔Consumer path outside the server. To compare with the Producer-side *Mean Send-to-Receive* in the secure scenario (17.40 ms, Table 7.1), we add the baseline Producer↔Consumer path measured without security (3.67 ms):

$$14.24\text{ ms} \;+\; 3.67\text{ ms} \;\approx\; 17.91\text{ ms} \;\approx\; 17.40\text{ ms}.$$

The small difference (∼0.5 ms, ∼3%) is consistent with run-to-run variance and the fact that the trace refers to a single transaction while Table 7.1 reports means over many. Overall, the packet-level evidence is consistent with the aggregated measurements and confirms that the secure scenario's overhead decomposes into: (i) the extra HTTP round trip for token *introspection* (Consumer↔Keycloak) and (ii) additional local processing (token result handling, Casbin evaluation, response encoding).

Bibliography

[1] Duncan Sparrell. *Open Command and Control (OpenC2) Language Specification Version 1.0.* Tech. rep. OASIS Open, 2022. URL: https://docs.oasis-open.org/openc2/oc2ls/v1.0/cs02/oc2ls-v1.0-cs02.html (cit. on pp. 2, 14).

[2] Duncan Sparrell. *Open Command and Control (OpenC2) Architecture Specification Version 1.0.* Tech. rep. OASIS Open, 2022. URL: https://docs.oasis-open.org/openc2/oc2arch/v1.0/cs01/oc2arch-v1.0-cs01.html (cit. on pp. 2, 12, 13, 36).

[3] D. Hardt. *The OAuth 2.0 Authorization Framework.* RFC 6749, Internet Engineering Task Force (IETF). 2012. URL: https://datatracker.ietf.org/doc/html/rfc6749 (cit. on pp. 3, 10, 11, 18, 19, 21, 23, 25–28, 35, 38, 39, 43).

[4] Matteo Repetto et al. «openc2lib: a Flexible, Portable, and Extensible Library for Remote Control of Security Functions». In: (2023) (cit. on pp. 3, 35, 49).

[5] Dieter Gollmann. *Authentication, Authorisation & Accountability Knowledge Area.* Tech. rep. v1.0.2. Accessed: October 2025. The Cyber Security Body Of Knowledge (CyBOK), 2021 (cit. on pp. 6, 9–11).

[6] Frontegg. *8 Access Control Types to Know in 2025.* https://frontegg.com/blog/access-control-types. Accessed: October 2025. Sept. 2025 (cit. on p. 7).

[7] Vincent C. Hu, David Ferraiolo, and D. Richard Kuhn. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations.* Tech. rep. NIST SP 800-162. National Institute of Standards and Technology, 2014. DOI: 10.6028/NIST.SP.800-162 (cit. on p. 7).

[8] MobiDev. *Access Control Security Models Explained: ACL vs RBAC vs ABAC.* https://mobidev.biz/blog/access-control-models-explained-acl-vs-rbac-vs-abac. Accessed: October 2025. Sept. 2025 (cit. on p. 8).

[9] Delinea. *Access Control Models and Methods | Types of Access Control*. https://delinea.com/blog/access-control-models-methods. Accessed: October 2025. 2023 (cit. on p. 8).

[10] Sumo Logic. *Discover authentication factors | 5 categories*. https://www.sumologic.com/glossary/authentication-factor. Accessed: October 2025. 2024 (cit. on p. 9).

[11] SecHard. *Most Common Types Of Password Attacks And How To Prevent Them*. https://sechard.com/blog/most-common-types-of-password-attacks-and-how-to-prevent-them/. Accessed: October 2025. 2023 (cit. on p. 10).

[12] Auth0. *JSON Web Token Introduction*. https://jwt.io/introduction. Accessed: October 2025. 2025 (cit. on p. 10).

[13] Cloudflare. *What is token-based authentication?* https://www.cloudflare.com/learning/access-management/token-based-authentication/. Accessed: October 2025. 2024 (cit. on p. 11).

[14] Office of the Victorian Information Commissioner. *Biometrics and Privacy*. Tech. rep. Accessed: October 2025. OVIC, 2019 (cit. on p. 11).

[15] Identity.com. *Privacy Concerns With Biometric Data Collection*. https://www.identity.com/privacy-concerns-with-biometric-data-collection/. Accessed: October 2025. Sept. 2025 (cit. on p. 11).

[16] Duncan Sparrell. *Open Command and Control (OpenC2) Profile for Stateless Packet Filtering Version 1.0*. Tech. rep. OASIS Open, 2022. URL: https://docs.oasis-open.org/openc2/oc2slpf/v1.0/cs01/oc2slpf-v1.0-cs01.html (cit. on p. 16).

[17] OASIS OpenC2 Technical Committee. *Specification for Transfer of OpenC2 Messages via HTTPS Version 1.1*. Tech. rep. Committee Specification 01. OASIS Standard, Nov. 2021. URL: https://docs.oasis-open.org/openc2/open-impl-https/v1.1/cs01/open-impl-https-v1.1-cs01.pdf (cit. on pp. 16, 37).

[18] Eran Hammer-Lahav. *The OAuth 1.0 Protocol*. Request for Comments 5849. Internet Engineering Task Force (IETF), Apr. 2010. URL: https://www.rfc-editor.org/rfc/rfc5849 (cit. on p. 19).

[19] Michael B. Jones, Dick Hardt, and David Recordon. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Request for Comments 6750. Internet Engineering Task Force (IETF), Oct. 2012. URL: https://www.rfc-editor.org/rfc/rfc6750 (cit. on pp. 19, 21, 51).

[20] *OAuth 2.0 Security Best Current Practice.* Request for Comments 9700. Internet Engineering Task Force (IETF), 2024. URL: https://www.rfc-editor.org/rfc/rfc9700 (cit. on pp. 19, 21).

[21] Michael B. Jones, Dick Hardt, and David Recordon. *Proof Key for Code Exchange by OAuth Public Clients.* Request for Comments 7636. Internet Engineering Task Force (IETF), Oct. 2012. URL: https://www.rfc-editor.org/rfc/rfc7636 (cit. on pp. 23, 45).

[22] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. *OAuth 2.0 Security Best Current Practice.* Internet-Draft draft-ietf-oauth-security-bcp-23. Work in Progress. Internet Engineering Task Force (IETF), Mar. 2021. URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-bcp-23 (cit. on pp. 23, 37).

[23] Casbin Authors. *Casbin: An Authorization Library that Supports Access Control Models like ACL, RBAC, ABAC.* Accessed: October 2025. 2020. URL: https://casbin.org (cit. on pp. 29, 35).

[24] Vincent C. Hu, David F. Ferraiolo, D. Richard Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. «Guide to Attribute Based Access Control (ABAC) Definition and Considerations». In: *NIST Special Publication 800-162* (2015). DOI: 10.6028/NIST.SP.800-162 (cit. on p. 29).

[25] Casbin Authors. *How It Works.* https://casbin.org/docs/how-it-works. 2024 (cit. on p. 30).

[26] Hsiaoming Yang. *Authlib: The ultimate Python library in building OAuth, OpenID Connect clients and servers.* https://authlib.org/. Accessed: October 2025. 2017 (cit. on pp. 35, 42).

[27] J. Richer. *OAuth 2.0 Token Introspection.* RFC 7662. IETF, Oct. 2015. URL: https://www.rfc-editor.org/info/rfc7662 (cit. on p. 43).

[28] Justin Richer, Michael B. Jones, John Bradley, Maciej Machulak, and Phil Hunt. *OAuth 2.0 Dynamic Client Registration Protocol.* RFC 7591. July 2015. DOI: 10.17487/RFC7591. URL: https://www.rfc-editor.org/info/rfc7591 (cit. on p. 44).

[29] Nicola Poidomani. *otupy Source Code: Authorizer.py.* Git Repository. File: src/otupy/auth/Authorizer.py. 2025 (cit. on p. 54).