



**Politecnico
di Torino**

Master of Science in Cybersecurity
Master's Degree Thesis

Extending OpenC2 for the Unified Control of Modern Security Functions

Supervisors

Prof. Fulvio Valenza

Prof. Daniele Brighenti

Prof. Matteo Repetto

Candidate

Dott. Hani Zeid

Academic Year 2025-2026

Summary

OpenC2 has been proposed as uniform language for remote command and control of security functions. It defines a common language and specific profiles for different classes of security functions (e.g., monitoring, firewalling, prevention, scanning).

Despite the great effort behind the language definition, it is still unclear how much a common language is able to abstract very different types of functions, since existing implementations are limited to simple cases only (i.e., stateless packet filter). This thesis investigates the usage of OpenC2 to control both legacy as well as more latest-generation of security functions in modern computing environments. The thesis includes design, implementation, and validation phases.

The design has reviewed existing OpenC2 drafts, that has been used/extended/re-designed to define the following profiles: a generic remote CLI profile that allows to load configuration files and start/stop security functions by invoking commands in a shell; file monitoring profile for configuring the collection of logs, either locally or in a centralized location; a network monitoring profile for configuring Netflow-compatible agents.

The implementation has leveraged the CNR's OpenC2 library (openc2lib) to develop a set of OpenC2 Actuators for the most common security functions used for network security. Finally, a number of test cases has been elaborated to validate the implementation with both “good” and “bad” conditions, including verification of formal correctness of the OpenC2 syntax. In addition, performance evaluation is also included in the thesis, focusing on CPU and memory usage as well as execution latency.

Table of Contents

List of Tables	IX
List of Figures	XI
1 Introduction	1
1.1 Cybersecurity challenges	1
1.2 Integration and Vendor Diversity Issues	1
1.3 Need for Standardized Interfaces	2
2 OpenC2:A Standardized Solution	3
2.1 Overview of OpenC2	3
2.2 OpenC2 System Architecture	3
2.2.1 OpenC2 Command	4
2.2.2 OpenC2 Response	4
2.3 Design Patterns	5
2.3.1 Consumer Device	5
2.4 OpenC2 Stack and Implementation	6
2.5 OpenC2 Actuator Profile	9
2.6 Conformance	9
3 OpenC2 Library	10
3.1 Overview of the OpenC2 Library	10
3.2 Architecture of openc2lib	11
3.3 Message Serialization and Deserialization	13
3.4 HTTP-Based Message Transmission and Reception	13
3.5 Utilizing openc2lib	14
3.5.1 Server-Side Implementation	15
3.5.2 Controller Development	15
3.5.3 Sending OpenC2 Commands	16
4 Thesis's Goal	18

5	Remote Command-Line Interface Actuator Profile	20
5.1	Objectives	20
5.1.1	Operational Workflow	21
5.2	Data Model	22
5.2.1	Database Structure	23
5.2.2	Artifact Tracking	23
5.2.3	Process Tracking	23
5.2.4	Mapping to OpenC2 Responses	23
5.2.5	Security Enforcement	24
5.3	Command Components	24
5.3.1	Actions	24
5.3.2	Targets	25
5.3.3	Arguments	25
5.3.4	Actuator Specifiers	26
5.4	Command Matrix	26
5.4.1	Features	26
5.4.2	Artifact	27
5.4.3	x-rcli:processes	27
5.4.4	x-rcli:files	27
5.5	Response Components	28
5.5.1	Response Status Code	28
5.5.2	Common Results	29
5.5.3	Specific Results	29
5.6	Data Types	30
5.6.1	Feature (Extended)	30
5.6.2	Process (Extended)	30
5.6.3	State	31
5.7	Implementation Details and Security Considerations	32
5.7.1	Overall structure and dispatching	32
5.7.2	Input validation and profile enforcement	32
5.7.3	Command execution model and hardening notes	32
5.7.4	Process lifecycle, authorization and persistence	33
5.7.5	File/artifact handling and integrity	34
5.7.6	Scheduling and timing model	34
5.7.7	Response normalization and logging	35
5.7.8	Implemented mitigations and residual risks	35
6	Network Flow Monitor	37
6.1	Objectives	37
6.1.1	Operational Workflow	37
6.2	Data Model	39

6.2.1	Database Structure	39
6.2.2	Monitor Tracking	39
6.2.3	Session Lifecycle	40
6.2.4	Mapping to OpenC2 Responses	40
6.2.5	Security Enforcement	40
6.3	Command Components	40
6.3.1	Actions	40
6.3.2	Targets	41
6.3.3	Arguments	41
6.3.4	Actuator Specifiers	42
6.4	Command Matrix	42
6.4.1	Features	43
6.4.2	x-nfm:monitor	43
6.4.3	x-nfm:monitor_id	43
6.5	Response Components	44
6.5.1	Response Status Code	44
6.5.2	Common Results	45
6.5.3	Specific Results	46
6.6	Data Types	47
6.6.1	Feature (Extended)	47
6.6.2	FlowMonitor	48
6.6.3	MonitorID	48
6.6.4	Exporter	48
6.6.5	Collector	49
6.6.6	ExportOptions	49
6.6.7	IE	50
6.6.8	Interface	50
6.6.9	IfaceType	51
6.6.10	FlowFormat	51
6.7	Implementation Details and Security Considerations	52
6.7.1	Architectural Overview	52
6.7.2	Backend Implementations	53
6.7.3	Process Management and Persistence	53
6.7.4	Input Handling and Runtime Configuration	53
6.7.5	Security Considerations	54
6.7.6	Observed Risks and Future Hardening	54
7	File Collection Log Monitoring	55
7.1	Objectives	55
7.1.1	Operational Workflow	55
7.2	Data Model	57

7.2.1	Database Structure	57
7.2.2	Monitor Tracking	57
7.2.3	Lifecycle Management	58
7.2.4	Security Enforcement	58
7.3	Command Components	58
7.3.1	Actions	58
7.3.2	Targets	59
7.3.3	Arguments	59
7.3.4	Actuator Specifiers	60
7.4	Command Matrix	60
7.4.1	Features	60
7.4.2	x-fclm:monitor	61
7.4.3	x-fclm:monitor_id	61
7.5	Response Components	62
7.5.1	Response Status Code	62
7.5.2	Common Results	63
7.5.3	Specific Results	63
7.6	Data Types	64
7.6.1	Feature (Extended)	64
7.6.2	Exporter	65
7.6.3	Collector	65
7.6.4	FileFormat	66
7.6.5	ImportOptions	66
7.6.6	EF (Export Field)	67
7.6.7	Socket	67
7.7	Implementation Details and Security Considerations	68
7.7.1	Implementation Overview	68
7.7.2	Execution Flow	68
7.7.3	Configuration and Process Control	69
7.7.4	Security Considerations	69
7.7.5	Cross-Cutting Security Controls	70
8	Validation and Testing	72
8.1	Overview	72
8.2	Data Type Validation	72
8.2.1	RCLI Profile	73
8.2.2	NFM Profile	73
8.2.3	FCLM Profile	74
8.2.4	Findings and Resolution	75
8.3	JSON Command and Response Validation	77
8.3.1	RCLI Profile	77

8.3.2	NFM Profile	78
8.3.3	FCLM Profile	80
8.3.4	Implementation and Interoperability Considerations	81
8.4	Performance Evaluation	82
8.4.1	RCLI Profile	82
8.4.2	NFM Profile	83
8.4.3	FCLM Profile	84
8.4.4	Measurement Context and Interpretation	84
9	Conclusion	85
	Appendices	86
A	OpenC2 Profile Command Examples	87
A.1	Remote Command-Line Interface Profile	87
A.1.1	Features (Query)	87
A.1.2	Artifact (Copy)	88
A.1.3	x-rcli:processes (Start, Stop, Query)	89
A.1.4	x-rcli:files (Query, Delete)	90
A.2	NFM Command and Response Examples	92
A.2.1	Features (Query)	92
A.2.2	x-nfm:monitor (Start)	93
A.2.3	x-nfm:monitor_id (Stop, Query)	94
A.3	File Collection and Log Monitoring Profile	96
A.3.1	Features (Query)	96
A.3.2	x-fclm:monitor (Start)	97
A.3.3	x-fclm:monitor_id (Stop)	99
	Bibliography	100

List of Tables

5.1	Mapping of example workflow steps to OpenC2 commands	21
5.2	RCLI Runtime Data Model (SQLite)	23
5.3	Actions Used in the RCLI Profile	24
5.4	Targets Used in the RCLI Profile	25
5.5	Arguments Used in the RCLI Profile	25
5.6	Argument Applicability Across Actions	25
5.7	Allowed Action–Target Pairs in the RCLI Profile	26
5.8	Fields Composing an OpenC2 Response in the RCLI Profile	28
5.9	RCLI Response Status Codes and Their Descriptions	28
5.10	Common OpenC2 Result Fields in the RCLI Context	29
5.11	RCLI Profile-Specific Result Fields	29
5.12	Extended Feature Enumeration Including the RCLI-Specific Entry clicommands	30
5.13	Extended Process Data Type Definition	31
5.14	State Enumerated Type Definition	31
6.1	Mapping of typical NFM workflow steps to OpenC2 commands	39
6.2	NFM Runtime Data Model (SQLite)	39
6.3	Actions Used in the NFM Profile	41
6.4	Targets Used in the NFM Profile	41
6.5	Arguments Used in the NFM Profile	41
6.6	Argument Applicability Across Actions	42
6.7	Actuator Specifiers Used in the NFM Profile	42
6.8	Allowed Action–Target Pairs in the NFM Profile	43
6.9	Fields Composing an OpenC2 Response in the NFM Profile	44
6.10	NFM Response Status Codes and Descriptions	45
6.11	Common OpenC2 Result Fields in the NFM Context	45
6.12	NFM Profile-Specific Result Fields	46
6.13	Extended Feature Enumeration for the NFM Profile	47
6.14	FlowMonitor Data Type Definition	48
6.15	MonitorID Data Type Definition	48

6.16	Exporter Data Type Definition	49
6.17	Collector Data Type Definition	49
6.18	ExportOptions Data Type Definition	50
6.19	Information Element (IE) Data Type Definition	50
6.20	Interface Data Type Definition	51
6.21	IfaceType Enumerated Type Definition	51
6.22	FlowFormat Enumerated Type Definition	52
7.1	Representative Command Flow for FCLM	57
7.2	FCLM Runtime Data Model (SQLite)	57
7.3	Actions Used in the FCLM Profile	58
7.4	Targets Used in the FCLM Profile	59
7.5	Arguments Used in the FCLM Profile	59
7.6	Argument Applicability Across FCLM Actions	59
7.7	Actuator Specifiers Used in the FCLM Profile	60
7.8	Allowed Action–Target Pairs in the FCLM Profile	60
7.9	Fields Composing an OpenC2 Response in the FCLM Profile	62
7.10	FCLM Response Status Codes	62
7.11	Common OpenC2 Result Fields in the FCLM Context	63
7.12	FCLM Profile-Specific Result Fields	63
7.13	Extended Feature Enumeration for the FCLM Profile	65
7.14	Exporter Data Type Definition	65
7.15	Collector Data Type Definition	66
7.16	FileFormat Enumerated Type Definition	66
7.17	ImportOptions Map Type Definition	66
7.18	EF String Wrapper Definition	67
7.19	Socket Record Type Definition	67
8.1	RCLI Data Type Validation Summary	73
8.2	NFM Data Type Validation Summary	74
8.3	FCLM Data Type Validation Summary	75
8.4	RCLI JSON Validation Summary (53 passed, 0 failed, 124.45s) . . .	78
8.5	NFM JSON Validation Summary (73 passed, 0 failed, 10.65s) . . .	80
8.6	FCLM JSON Validation Summary (53 passed, 0 failed, 9.84s) . . .	81
8.7	RCLI Controller-side latency (seconds)	82
8.8	RCLI Server-side latency (seconds)	83
8.9	NFM Controller-side latency (seconds)	83
8.10	NFM Server-side latency (seconds)	83
8.11	FCLM Controller-side latency (seconds)	84
8.12	FCLM Server-side latency (seconds)	84

List of Figures

2.1	OpenC2 Message Exchange	4
2.2	Actuator Manager Device	6
2.3	OpenC2 Layering Model	6
3.1	OpenC2 library architecture	11
5.1	Example of a typical command exchange workflow between OpenC2 Producer and RCLI Consumer	22
6.1	Typical operational workflow of the NFM actuator profile	38
7.1	Typical operational workflow of the FCLM actuator profile	56

Chapter 1

Introduction

1.1 Cybersecurity challenges

The attack surface of cyber systems continues to expand as connectivity, functionality, and system complexity increase. Meanwhile, adversaries have become more capable, leveraging advanced tactics, automation, and increased operational resources. The consequences of cyberattacks include financial losses, intellectual property theft, operational disruption, reputational damage, national security risks, and wider psychological and societal impacts.

However, many cybersecurity environments remain statically configured and operate in isolation, making defensive actions slow and fragmented. This creates an inherent time disadvantage for defenders, especially in large-scale and coordinated attacks that unfold at machine speed [1]. The need for more adaptive, automated, and integrated cyber defense has been recognized in updated cybersecurity frameworks such as NIST CSF 2.0, which emphasizes the importance of machine-speed response mechanisms [2].

1.2 Integration and Vendor Diversity Issues

Modern infrastructures consist of tightly interconnected devices, applications, and services deployed across both physical and virtual domains. As a result, defense mechanisms must operate in a coordinated manner to prevent adversaries from exploiting inconsistencies across the security environment. Research in security orchestration and dynamic service composition has explored approaches to unify detection and response workflows.

In practice, however, organizations frequently operate hundreds of heterogeneous security tools, each with its own proprietary configuration methods. While such diversity can increase system resilience, it complicates the translation and

enforcement of consistent security policies across environments. Studies have shown that the lack of interoperability and standardized action semantics significantly limits the scalability of defense orchestration solutions [3, 4]. These challenges have motivated efforts to establish higher-level, vendor-agnostic control models to support automated security operations.

1.3 Need for Standardized Interfaces

Standardized function-centric interfaces for cyber defense command and control can simplify integration, reduce operational burden, and improve response agility. Coordinated cyber defense requires reliable machine-to-machine communication, enabling systems to execute defensive actions at operational speed. Decoupling system functionality and defining common control semantics promotes interoperability and supports gradual technology evolution.

A relevant example is the process of updating firewall rules across multiple geographically distributed devices during an ongoing incident. Without standardization, this requires manual access to individual devices or reliance on vendor-specific APIs. By contrast, an open and platform-agnostic command language allows a single, abstract command to be issued, with each device translating it into its native configuration format [1].

Open Command and Control (OpenC2) addresses this need by defining a standardized language for cyber defense actions that is independent of underlying vendor or device implementations. Originating from the Integrated Adaptive Cyber Defense (IACD) framework [5], OpenC2 enables automated and coordinated defensive responses. It has been applied in cloud environments to orchestrate firewalls and intrusion detection components [6] and has been shown to integrate effectively with threat intelligence workflows based on STIX/TAXII [7]. The overall aim is to support interoperable, machine-speed cyber defense across diverse operational environments.

Chapter 2

OpenC2: A Standardized Solution

2.1 Overview of OpenC2

Open Command and Control (OpenC2) is a standardized and extensible language that allows machine-to-machine communication for the command and control of cyber defense components, subsystems, and/or systems in a manner that is agnostic of the underlying products, technologies, transport mechanisms, or other aspects of the implementation. This language, used in conjunction with OpenC2 Actuator Profiles [8] and OpenC2 Transfer Specifications [9][10], allows vendor-agnostic cybertime response to attacks. The goal of OpenC2 is to enable coordinated defense in cyber-relevant time between decoupled blocks that perform cyber-defense functions.

The OpenC2 language [11] assumes that the event has been detected, a decision to act has been made, the act is warranted, and the initiator and recipient of the Commands are authenticated and authorized. The OpenC2 language was designed to be agnostic of the other aspects of cyber defense implementations that realize these assumptions.

2.2 OpenC2 System Architecture

The OpenC2 Language Specification [11] defines a language used to compose messages for command and control of cyber defense systems and components. These messages are bidirectional and are exchanged between Producer and Consumer(s). A Message consists of a header and a payload (defined as a Message body). The language defines two payload structures: Command and Response

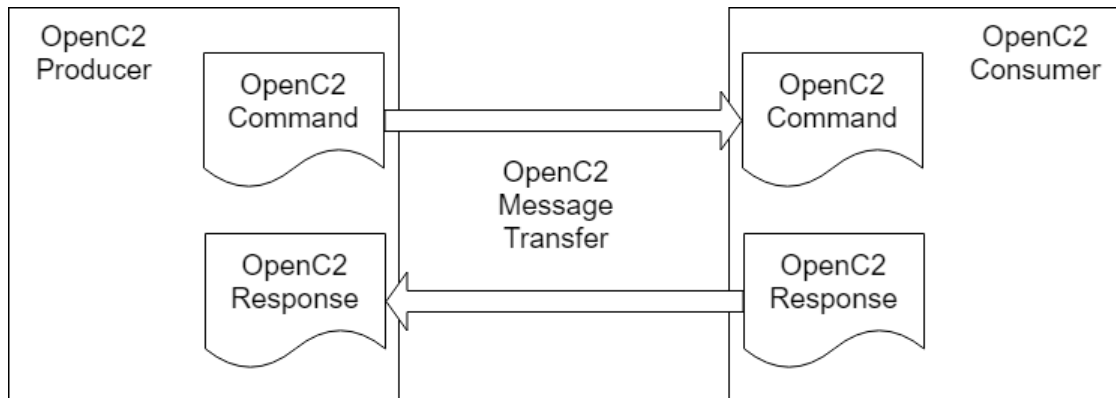


Figure 2.1: OpenC2 Message Exchange

2.2.1 OpenC2 Command

An instruction from one system known as the Producer, to one or more systems, the Consumer(s), to act on the content of the Command. OpenC2 Command is composed of four main fields:

- Action (required): the operation to be executed by an actuator.
- Target (required): the object upon which the action is executed.
- Arguments (optional): extra parameters to provide details how, when or where to execute a command.
- Actuator (optional): the entity responsible for executing the action.
 - Specifier: A field or attribute that provides a Target's precise identification.

2.2.2 OpenC2 Response

Any information sent back to the Producer as a result of the Command. The Response provides information about the results of executing a Command. It is composed of three fields:

- Status (required): integer status code.
- Status text (optional): description of the Response status.
- Results (optional): results derived from the executed Command.

Command fields follow common language patterns, defining a subject:Actuator, a verb:Action, an object:Target, and complements:Arguments, with at least a single Action-Target pair. OpenC2 Language Specification defines a set of limited actions, with no extension or creation of new ones allowed, to ensure compatibility with diverse systems.

It is possible to define new Targets other than the ones in the specification by specifying a new Actuator Profile. Actuator Profiles specify semantic constraints and allow extension of the language for specific security functions. This is necessary since not every action will be appropriate for all use cases, and certain limits must be set in order to effectively accurately model real-world circumstances. The profile under which the Action will be performed is identified by the Actuator field.

2.3 Design Patterns

Design patterns are reusable solutions to common design problems, especially in complex systems. They are the secret to creating scalable, maintainable, and interoperable solutions. Design patterns in OpenC2 are a vital element in specifying how command and control systems can engage and interact best. The use of OpenC2 design patterns is what this thesis focuses on, with specific interest in OpenC2 Consumer device configurations and with special attention to one of the specific types of configuration.

2.3.1 Consumer Device

According to OpenC2 Architecture [12] there are four representative configurations for an OpenC2 Consumer device:

- The Consumer implements a single function, and therefore supports a Actuator Profile (AP).
- The Consumer implements several functions, and consequently supports multiple APs.
- The Consumer is a manager for a collection of devices and the interface to them is not assumed to use OpenC2.
- The Consumer is a manager for a collection of devices that are managed using OpenC2.

The profiles implemented in this thesis mainly focuses on the third configuration where the Consumer is a manager of collection of devices who is giving the producer the possibility to communicate with the managed devices through a proxy

(Consumer). The Consumer here is translating the Producer's commands to the managed devices which could be inside host or an external source (e.g. in the cloud).

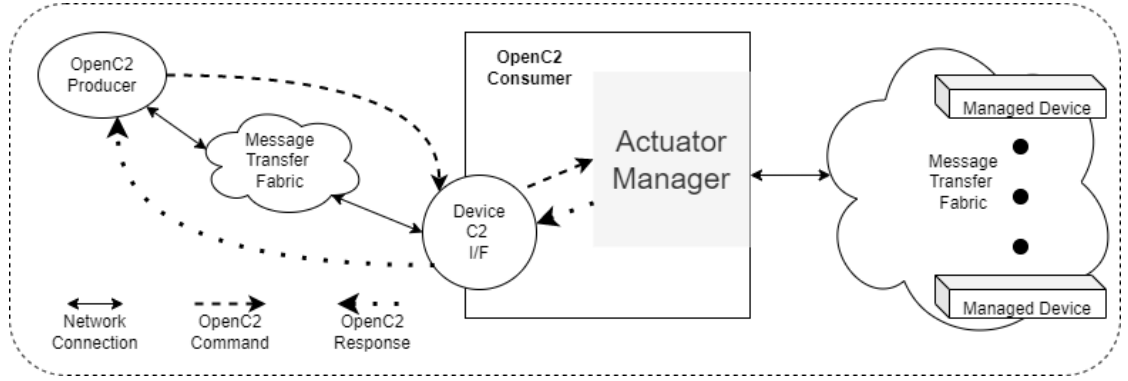


Figure 2.2: Actuator Manager Device

2.4 OpenC2 Stack and Implementation

In order to facilitate smooth interactions across cybersecurity components, the OpenC2 language can be used in a layered architecture where many standards and protocols cooperate. The OpenC2 Actuator Profiles [8] outline the commands and responses relevant to specific security functions, while the OpenC2 Language Specification [11] provides the language's basic constructions.

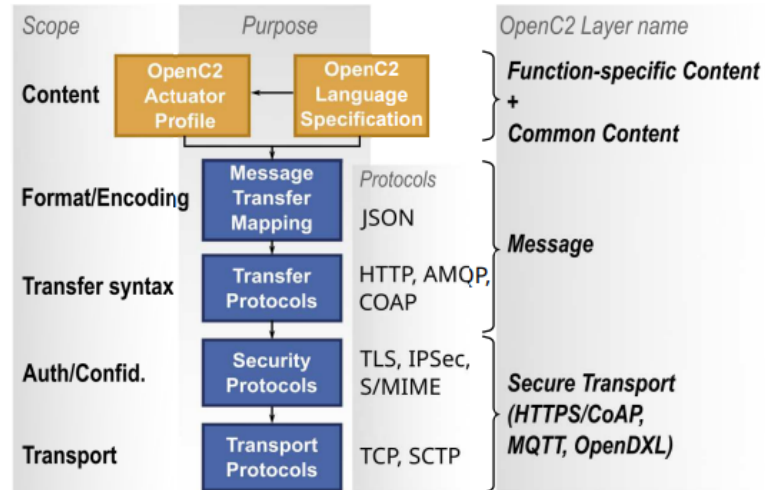


Figure 2.3: OpenC2 Layering Model

JSON is used by default for message encoding, even though OpenC2 does not specify a format. For correct tracking and execution, OpenC2 messages usually include the following information: content type, version, sender, timestamp, and correlation ID. Security in the form of TLS, IPSEC, or S/MIME offers encryption, authentication, and integrity protection, while Transfer Specifications provide the protocols, such as HTTPS [9] or MQTT [10], that are used to transmit OpenC2 communications.

For instance, when using HTTPS, an OpenC2 Producer first connects to a Consumer using TCP, then encrypts the communication by creating a TLS session. After that, HTTP POST operations are used to send OpenC2 instructions and responses with a consistent URI path (`/.well-known/openc2`). This promotes uniformity and compatibility among implementations. Effective tracking of submitted instructions and their results is made possible by the inclusion of a distinct X-Request-ID in both the HTTP and OpenC2 headers to match responses to requests.

An example of an HTTP message containing an OpenC2 Command is shown in the code below. According to the General Requirements of the HTTP specification, the POST method is always used. The URI is a `/.well-known/` path as well. The HTTP message body contains OpenC2 messages in JSON format, which are followed by OpenC2 version 1.0, as indicated by the "Content-type" field. Commands and Responses are uniquely identified via the "X-Request-ID" extension header, which enables activity correlation between them. A unique identifier must be included by the Producer in either the "request_id" OpenC2 message header or the "XRequest-ID" HTTP header. Each instruction and its matching response can be connected in this manner. The original "request_id" value from the OpenC2 Command's header is replicated into the HTTP header. The OpenC2 Command's body is its content. The same Command fields are present in the Response message's HTTP and OpenC2 headers.

```
POST /.well-known/openc2 HTTP/1.1
Content-type: application/openc2+json;version=1.0
Date: Wed, 19 Dec 2018 22:15:00 GMT
X-Request-ID: d1ac0489-ed51-4345-9175-f3078f30afe5
{
  "headers": {
    "request_id": "d1ac0489-ed51-4345-9175-f3078f30afe5"
  },
  "created": 1545257700000,
  "from": "oc2producer.company.net",
  "to": [
```

```
        "oc2consumer.company.net"
    ],
    },
    "body": {
        "openc2": {
            "request": {
                "action": "...",
                "target": "...",
                "args": "..."
            }
        }
    }
}
```

Listing 2.1: HTTP Message Containing an OpenC2 Command

```
Content-type: application/openc2+json;version=1.0
Date: Wed, 19 Dec 2018 22:15:00 GMT
X-Request-ID: d1ac0489-ed51-4345-9175-f3078f30afe5
{
    "headers": {
        "request_id": "d1ac0489-ed51-4345-9175-f3078f30afe5"
    },
    "created": 1545257700000,
    "from": "oc2producer.company.net",
    "to": [
        "oc2consumer.company.net"
    ]
},
    "body": {
        "openc2": {
            "request": {
                "action": "...",
                "target": "...",
                "args": "..."
            }
        }
    }
}
```

Listing 2.2: HTTP Message Containing an OpenC2 Response

2.5 OpenC2 Actuator Profile

Interoperability amongst security solutions is made possible by an OpenC2 Actuator Profile [8], which gives cybersecurity tools a common method for interpreting and executing OpenC2 commands. For a variety of security functions, including firewall, intrusion detection systems (IDS), endpoint protection, and malware mitigation, the profiles specify the precise commands, inputs, answers, and execution limitations.

In order for various devices from various suppliers to read and perform the same instruction in a comparable manner, each actuator profile converts the OpenC2 language into a particular security function. To ensure that firewalls from various manufacturers react similarly, an actuator profile for stateless packet filtering, for instance, specifies accept, deny, or drop orders for network traffic.

Actuator profiles boost automation, decrease complexity, and speed up response to cyberattacks by mandating normalized behavior. Security teams can plan defenses more effectively and quickly thanks to actuator profiles, which also reduce the need for specialized integrations and communication channels.

2.6 Conformance

The conformance clauses in the specifications, which outline the proper usage of the language, must be followed by all OpenC2 communications.

Chapter 3

OpenC2 Library

3.1 Overview of the OpenC2 Library

A library named “openc2lib” was developed to manage the grammar and protocols defined by the openc2 specification, described by the OpenC2 standards, and it was developed in Python due to this language’s ability to be effortlessly utilized in various software environments. Additionally, Python accommodates reflective programming paradigms, allowing the examination, introspection, and alteration of its own composition and actions during execution, thus allowing the user to create new profiles, transport protocols, and encoding formats while maintaining the essence of the library.

The aim of the library is to establish a flexible stack for moving Commands and Responses as outlined in OpenC2 specifications. Using the library, it is feasible to execute both Consumers and Producers and establish the complete stack for each one. Additionally, the library offers a versatile and expandable system for serialization utilizing Python dictionaries as intermediary data representation. Python dictionaries are organized and mutable collections that prohibit duplicates and are used for storing data values in key:value pairs. Using the intermediary lexicon representation, OpenC2 entities can be serialized without having to develop custom and repetitive techniques for each one, as the library can be readily expanded to include serialization formats like XML, YAML, JSON and CBOR.

The complete structure and operation of the OpenC2 library are illustrated in the figure below, where one can observe that the encoding process of the Producer is identical to the decoding process of the Consumer. The JSONEncoder allows the serialization of messages in JSON format, while HTTPSTransfer enables message transfer through the HTTPS protocol. Furthermore, the Consumer is able to select the suitable actuator specified in the Command that has been received.

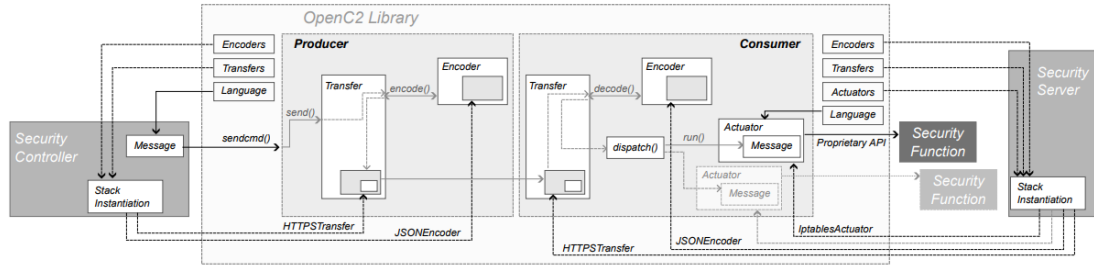


Figure 3.1: OpenC2 library architecture

3.2 Architecture of openc2lib

The several modules that make up the openc2lib architecture each implement a distinct aspect of the specifications.

- The “types” folder, which has three subfolders, contains the implementation of data types and structures:
 - **Base:** Language Specification [11] (section 3.1.1) base kinds (structures). Serialization processes are impacted by the requirement that each OpenC2 object derive from these classes. The implementations of the Choice, ArrayOf, Record, and other OpenC2 structural classes are included in this folder. To convert an object into a Python dictionary and vice versa, each base type has to implement the `todict` and `fromdict` methods.
 - **Data:** Section 3.4.2 of the Language Specification [11] defines the data types. The specification’s definitions serve as the basis for the class names. `IPv4Addr`, `L4Protocol`, and `Nsid` are a few examples of data types. For a data type class to be automatically serialized, it must be a child class that inherits all of the properties and methods of a parent class, which is a base type.
 - **Targets:** Targets listed in section 3.4.1 of the Language Specification [11]. All of a base type’s methods and properties must be inherited by a target class, which must be a child class. The class `Features` is an example of a target; it is defined as `ArrayOf(Feature)`, where `Feature` is the data type and `ArrayOf` is the base type.
- The classes that are helpful for implementing the OpenC2 standard language beginning with the previously established data are contained in the “core” subdirectory. Depending on their purpose, the modules can be separated into:
 - **Actions:** It lists the Actions specified in section 3.3.1.1 of the Language Specification [11].

- **Actuator:** This is the base class for deriving Actuators.
 - **Args:** It describes the Command Arguments as outlined in section 3.3.1.4 of the Language Specification [11]. Since additional instances can be added while implementing new profiles, it is organized as a map.
 - **Command:** Section 3.2 of the Language Specification [11] defines the OpenC2 Command structure, which is described in this module.
 - **StatusCode:** It shows the result of command execution.
 - **Results:** It establishes the fundamental framework for the results presented in a response. By adding more fields to profile definitions, this class can be expanded. It applies the definition of Result found in section 3.3.2.2 of the Language Specification [11].
 - **TargetRegister:** All available Targets, both from Profiles and the openc2lib, are registered by this class.
 - **Profile:** It is the Profile idea interpreted by openc2lib. It specifies the language extensions designated for that message as well as a Profile namespace. When it comes to controlling particular security functions, a profile is completely transparent to concrete implementation.
 - **Register:** This class registers every element that is available, whether it is from Profiles or the openc2lib. To enable the core system to use their classes and names for encoding and decoding, profiles can be filled in with extra definitions.
- The second group defines the Producer and the Consumer:
 - **Producer:** An “Encoder” and a “Transfer” protocol are utilized to build an OpenC2 stack. The function `sendcmd` is implemented, which generates the ‘Message’ metadata that will be encoded and transmitted internally and transmits an openc2lib command.
 - **Consumer:** It demonstrates how an OpenC2 Consumer server should behave while sending OpenC2 commands to actuators. In order to process a message, it builds the OpenC2 stack. While the function `dispatch` picks one or more Actuators to handle the Command after scanning the Actuator Profile carried in the Command, the function `run` enables the Consumer to receive messages in the network.
 - The third group defines how a message must be transmitted:
 - **Encoder:** Base encoding routines are provided by this module to convert openc2lib objects into a dictionary-based representation and vice versa.

- **Transfer:** The fundamental behavior of the Transfer protocols is defined by the interface.
- **Message:** According to section 3.2 of the Language Specification [11], it specifies the OpenC2 Message structure.

3.3 Message Serialization and Deserialization

A command or response needs to be encoded or decoded into a serialization format once it is prepared. OpenC2 communications can be encoded in a variety of forms, however JSON is the most used. The environment in which OpenC2 is used, together with the strengths and weaknesses of the chosen transfer protocol, all have an impact on the encoder selection.

Because OpenC2 defines its own data types and structures in the Language Specification [11], Python dictionaries are unable to serialize all OpenC2 data directly. To get around this problem, two functions for the base types must be defined: one that creates a dictionary from an OpenC2 element, and another that creates OpenC2 elements from a dictionary.

Although it works in a different way, the deserialization process is comparable to the serialization process. The JSON message to be decoded and the `openc2lib` class to convert it into are inputs. Deserializing the JSON message first creates a dictionary, which is subsequently converted into the `openc2lib` class.

Because they repeatedly invoke the Encoder class until the item is properly converted, the deserialization and serialization procedures operate recursively if the object cannot be turned straight into a dictionary.

3.4 HTTP-Based Message Transmission and Reception

After an OpenC2 Command has been encoded using the appropriate serialization format, it must be transmitted to the Consumer. Message transmission is based on the *Transfer* class, a common module that provides the fundamental methods for sending and receiving messages. Although Python does not explicitly support the concept of virtual classes or interfaces, the *Transfer* class plays an equivalent role by defining a basic template that must be implemented by derived classes. For each supported communication protocol, a corresponding child class must be implemented, inheriting the properties and functions of the *Transfer* class while specifying the appropriate message format and transmission logic.

For HTTP-based communication, this functionality is implemented by the *HTTPTransfer* class, which provides an interface for both HTTP and HTTPS protocols. The implementation is based on the Flask framework, which facilitates HTTP communication and simplifies testing and development.

Once encoded, the OpenC2 Command is inserted into the body of a new transfer message created using the *Message* class from the `transfers/http` module. This class defines the structure of a valid HTTP message. An important element of the message header is the `request_id`, which serves as a unique identifier generated by the Producer. This identifier is preserved in every Response issued by the Consumer, allowing the Producer to reference a specific Command, transaction, or event chain.

The HTTP message header is then constructed in compliance with Section 3.2.3 of the HTTPS Specification [9]. The message is transmitted using the *HTTPTransfer* class, and upon arrival at the Consumer, the appropriate decoding procedures are applied. This process begins with parsing the JSON content of the received message.

The Consumer determines which Actuator is responsible for processing the decoded OpenC2 Command by evaluating the Profile namespace identifier. The Action and Arguments contained in the Command are then forwarded to the selected Actuator in a standardized and predictable manner, independent of the internal implementation details of the security function. After executing the requested operation, the Actuator generates a Response, which is serialized using the Encoder module and transmitted back to the Producer through the Transfer module.

3.5 Utilizing openc2lib

Up until now, we have talked about the Python library's technical features. The implementation procedure is now the main focus. To create the `openc2lib` library, two entities are introduced: **Server** and **Controller**. The former mimics the actions of the OpenC2 Consumer, which might be an Internet of Things gadget or a firewall. The latter mimics the OpenC2 Producer, which might be a SOAR that sends commands to different network consumers, for instance. The Server and Controller implementations will be defined in the upcoming sections in order to make use of the `openc2lib` and start OpenC2 communication. Additionally, the final section reports an example of transmitting an OpenC2 Command.

3.5.1 Server-Side Implementation

The OpenC2 Consumer is designed to be instantiated and operated by a server. Before the Consumer can be created, the protocol stack and the set of actuators to be exposed must be properly configured. In addition to the list of available actuators and a string identifying the Consumer instance, two fundamental components are required: an Encoder and a Transfer protocol. The Encoder is responsible for translating OpenC2 messages into a concrete serialization format, such as JSON, while the Transfer component handles the actual transmission and reception of those messages over a specific communication channel.

When the Consumer object is instantiated, these elements are provided to its constructor. Both the Encoder and the Transfer parameters require concrete implementations of their respective interfaces—such as JSONEncoder for message encoding and HTTPTransfer for communication—rather than abstract base classes. To enable the server to receive network messages, the Consumer’s entry-point method, `run`, must be invoked.

In the code example shown below, a server is created with an actuator implementing the SLPF Profile and configured to listen on an HTTP channel at a specified IP address and port, in this case using the loopback interface.

```
import openc2lib as oc2
from openc2lib.encoders.json_encoder import JSONEncoder
from openc2lib.transfers.http_transfer import HTTPTransfer
import openc2lib.profiles.slpf as slpf
from openc2lib.actuators.iptables_actuator import IptablesActuator

actuators = {}
actuators[(slpf.Profile.nsid, 'iptables')] = IptablesActuator()

c = oc2.Consumer("consumer.example.net",
                 actuators,
                 JSONEncoder(),
                 HTTPTransfer("127.0.0.1", 8080))

c.run()
```

3.5.2 Controller Development

The purpose of a Controller is to instantiate an OpenC2 Producer and utilize it to manage a remote security feature. It starts with a constructor that accepts three parameters: an instance of an Encoding class implementing the base Encoder interface, an instance of a Transfer protocol implementing the base Transfer interface,

and a string identifying the Producer. Similar to the Server, the constructor for both the Encoder and the Transfer requires concrete implementations, such as HTTPTransfer and JSONEncoder, rather than the abstract base classes.

An example of properly initializing an openc2lib Producer that sends JSON-encoded messages to an openc2lib Consumer at a given IP address and port is provided below:

```
import openc2lib as oc2
from openc2lib.encoders.json import JSONEncoder
from openc2lib.transfers.http import HTTPTransfer

p = oc2.Producer("producer.example.net",
                 JSONEncoder(),
                 HTTPTransfer("127.0.0.1", 8080))
```

3.5.3 Sending OpenC2 Commands

The OpenC2 Command can be defined with all of its fields after the Controller has been defined. The following is a definition of the command arguments that will be used in the command:

```
arg = oc2.Args({'response_requested': oc2.ResponseType.complete})
```

The following command shows that the Producer wishes to use a query action to find out which OpenC2 versions and profiles the Consumer supports. Keep in mind that no actuators are mentioned:

```
cmd = oc2.Command(oc2.Actions.query,
                  oc2.Features([oc2.Feature.versions, oc2.Feature.
    profiles]),
                  arg,
                  actuator=None)
```

The selected Actuator Profile in this additional example is the SLPF with host-name "firewall," which will permit inbound traffic for the designated range of IPv4 addresses. The SLPF Profile is required to carry out this command on the consumer's end. The Controller then uses the function `sendcmd` to send the Command, returning the Response.

```
import openc2lib.profiles.slpf as slpf
```

```
pf = slpf.Specifiers({'hostname': 'firewall'})
arg = slpf.Args({'response_requested': oc2.ResponseType.complete,
                'direction': slpf.Direction.ingress})

cmd = oc2.Command(oc2.Actions.allow,
                  oc2.IPv4Net("172.19.0.0/24"),
                  arg, actuator=pf)

resp = p.sendcmd(cmd)
```

Chapter 4

Thesis's Goal

This thesis's primary goal is to investigate and assess OpenC2's potential as a standardized language for remotely managing security features in contemporary computer systems. In order to manage both historical and next-generation security features in a variety of network contexts, this thesis specifically attempts to design and implement new OpenC2 profiles. By presenting more intricate profiles that meet contemporary security requirements, the thesis will overcome the shortcomings of current OpenC2 implementations, which have mostly concentrated on straightforward use cases.

The key objectives include:

1. Designing a set of OpenC2 profiles, including:
 - A **generic remote CLI profile** for managing security functions via shell commands (e.g., load configuration files, start/stop services).
 - A **file monitoring profile** for configuring log collection and forwarding to a centralized location.
 - A **network monitoring profile** for configuring NetFlow-compatible agents to monitor traffic.
2. Implementing the designed profiles using the CNR's OpenC2 library (`openc2lib`) to develop Actuators for common network security functions.
3. Validating the implementation through rigorous testing, including:
 - Formal correctness checks of OpenC2 syntax.
 - Evaluation under both "good" and "bad" conditions to simulate real-world scenarios and error handling.
 - Performance assessment in terms of CPU and memory usage, and execution latency.

The ultimate objective is to show how OpenC2 may offer a uniform method for managing a broad range of security functions, enhancing adaptability and interoperability in dynamic security contexts.

Chapter 5

Remote Command-Line Interface Actuator Profile

5.1 Objectives

The Remote Command-Line Interface (RCLI) Actuator Profile aims to enable remote configuration and control of security functions through command-line interfaces. It is primarily conceived as a fallback mechanism for security components that do not natively implement any specific OpenC2 actuator profile, or for which a dedicated profile has not yet been defined. By providing a standardized method to issue commands remotely, the RCLI profile facilitates interaction with a wide range of security tools, allowing consistent management and control of both legacy and modern systems through a common command-line based interface.

The primary objective of the RCLI profile is to abstract the process of loading configuration files, starting and stopping security functions, and interacting with the system's shell environment, regardless of the underlying platform or specific implementation of the security tool. At the same time, the profile is designed to achieve this objective without introducing additional vulnerabilities into the system, ensuring that remote command execution is performed in a controlled and secure manner. By defining a common OpenC2-based language, the RCLI profile enables seamless and uniform communication with different security functions across heterogeneous environments while preserving the integrity and security of the managed systems.

This Actuator Profile is designed to allow Producers to send commands to Consumers (security tools) in a standardized way. The Consumer, upon receiving these commands, will process them and return appropriate responses based on the outcome. These responses provide feedback on the success or failure of the executed command, along with additional context or errors if applicable.

The RCLI Profile is implemented on the Consumer side, and it is one of the possible Actuator Profiles that the Consumer can support. Communication between Producer and Consumer follows the OpenC2 standard, where the Producer sends a command specifying that the Actuator to execute is RCLI. If the Consumer implements the RCLI profile, it will return a Response. If the Consumer does not support the RCLI profile, it will instead return an appropriate error message indicating that the requested actuator is not available.

5.1.1 Operational Workflow

This section provides a high-level description of how the Remote Command-Line Interface (RCLI) actuator profile is intended to be used in practice. The purpose of this overview is to give the reader an intuitive understanding of the interaction model before introducing the more detailed technical elements of the profile.

The RCLI profile enables a Producer to remotely manage command-line based security functions through a standardized set of OpenC2 commands. A typical interaction generally follows a logical progression: the Producer first discovers the capabilities of the Consumer, then optionally transfers files required for execution, starts one or more processes, monitors their status, and eventually stops them. If temporary artifacts were uploaded, they can be removed once they are no longer needed. This approach allows different security tools to be orchestrated in a consistent and technology-agnostic manner.

It should be noted that the use of the RCLI profile is not restricted to a fixed sequence of operations. Depending on the use case, a Producer may issue only a subset of the available commands. For example, a Producer might simply query running processes without uploading files, or delete previously uploaded artifacts without starting any new process. The workflow presented here therefore illustrates a representative and complete interaction, but not a mandatory one. In real deployments, Producers are free to combine the available actions in the way that best fits their operational requirements.

Step	Action	Target	Purpose
1	query	features	Discover supported capabilities
2	copy	artifact	Upload required file
3	start	x-rcli:processes	Launch CLI command
4	query	x-rcli:processes	Monitor execution
5	stop	x-rcli:processes	Terminate execution
6	delete	x-rcli:files	Cleanup uploaded resources

Table 5.1: Mapping of example workflow steps to OpenC2 commands

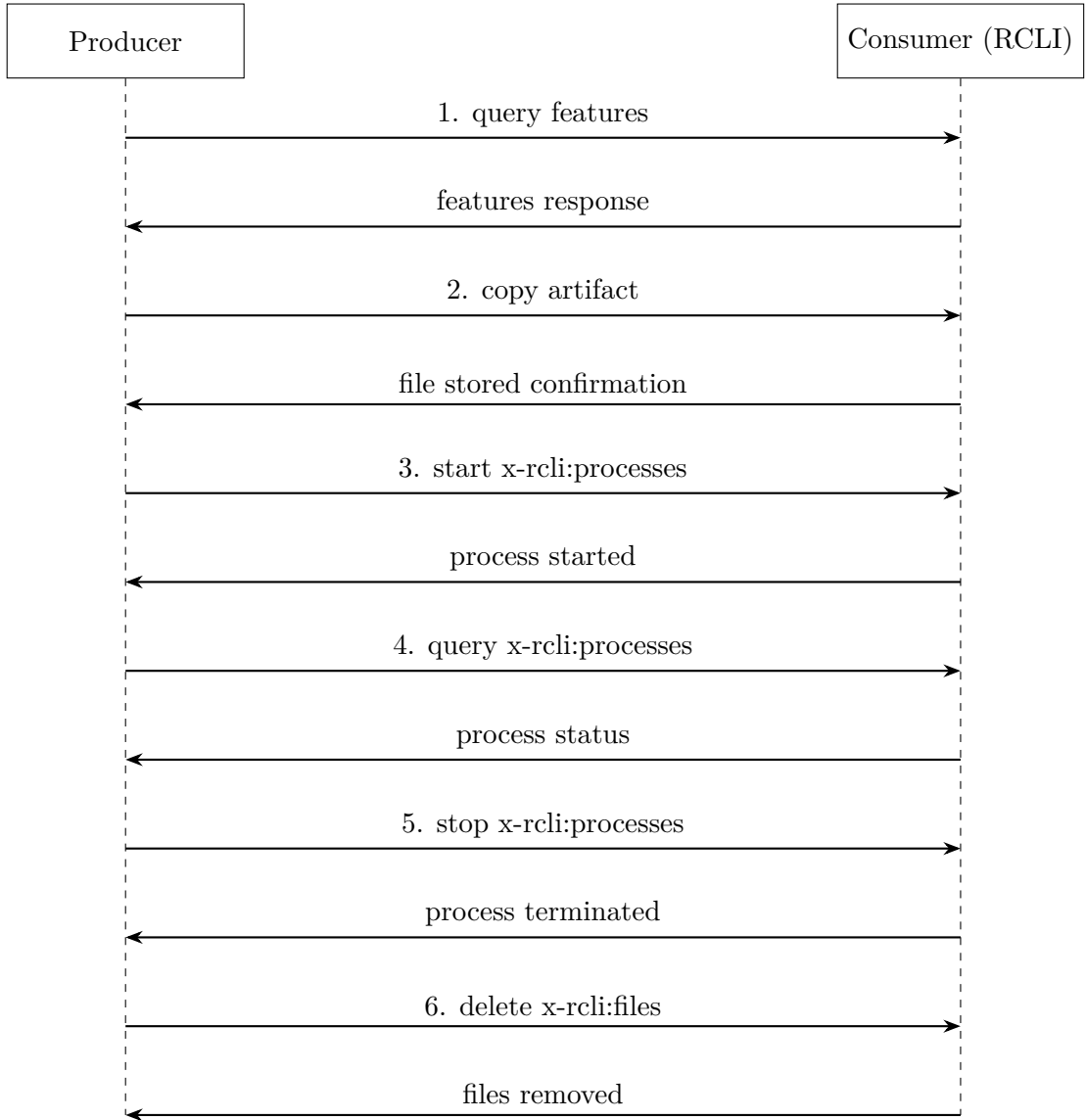


Figure 5.1: Example of a typical command exchange workflow between OpenC2 Producer and RCLI Consumer

5.2 Data Model

The **RCLI Profile** maintains a small runtime state to support file management and process control, and to ensure that these operations are secured on a per-producer basis. This state is stored in a local SQLite database alongside the actuator. The database is not used for long-term auditing; instead, it provides temporary ownership and authorization context for OpenC2 commands executed via RCLI.

5.2.1 Database Structure

The database consists of two primary tables shown in Table 5.2. Each table associates resources with the `uid` of the OpenC2 producer that created them, allowing the actuator to enforce ownership and prevent cross-tenant interference.

Table	Fields
CopiedArtifacts	<code>uid</code> , <code>path</code> , <code>name</code> , <code>hashes</code>
RunningProcesses	<code>uid</code> , <code>pid</code> , <code>name</code>

Table 5.2: RCLI Runtime Data Model (SQLite)

5.2.2 Artifact Tracking

The **CopiedArtifacts** table records files uploaded to the target system using the `copy` action. Each stored file is associated with the identity of the producer that uploaded it. Optional hash values are recorded when available, enabling integrity verification upon retrieval or execution. This mechanism ensures that a producer cannot access or delete files uploaded by another producer, reinforcing tenant isolation.

5.2.3 Process Tracking

The **RunningProcesses** table stores the processes that were started using the `start` action. Recording both the process identifier (`pid`) and `uid` allows the actuator to enforce that:

- A producer may only stop processes that it previously started.
- System processes or other unrelated processes are not exposed or modifiable through RCLI.

Entries are removed when a process terminates; the model is intentionally non-historical to remain lightweight and aligned with transient remote execution semantics.

5.2.4 Mapping to OpenC2 Responses

When returning information to a producer, the actuator derives OpenC2 *Results* objects directly from these tables:

- `file_status` arrays are generated from **CopiedArtifacts** rows.

- `process_status` arrays are generated from **RunningProcesses** rows.

This ensures consistent representation of runtime state across command responses and actuator implementations.

5.2.5 Security Enforcement

By binding each file and process to the `uid` of the producer that created it, the actuator enforces two fundamental security guarantees:

1. **File isolation:** A producer may retrieve or delete only the files it uploaded.
2. **Process ownership:** A producer may stop only processes it initiated.

It should be noted, however, that this mechanism assumes the `uid` of the producer to be trustworthy. An attacker able to spoof the `uid` of a legitimate producer could potentially bypass these protections. To mitigate this limitation, stronger authentication mechanisms can be integrated; for instance, recent work has proposed the adoption of OAuth2 within OpenC2 to provide more robust identity verification and prevent impersonation attacks.

5.3 Command Components

This section describes the OpenC2 Command elements relevant to the RCLI Profile: **Actions**, **Targets**, **Arguments**, and optional **Actuator Specifiers**. While the RCLI Profile reuses Actions from the Language Specification, it introduces new Targets and a profile-specific Argument.

5.3.1 Actions

The RCLI Profile uses the following Actions:

ID	Name	Description
3	query	Request information or capabilities.
9	start	Launch one or more processes.
10	stop	Terminate previously started processes.
20	delete	Remove a stored file.
28	copy	Transfer a file to the consumer.

Table 5.3: Actions Used in the RCLI Profile

5.3.2 Targets

The RCLI Profile introduces two new Targets and reuses standard ones:

Name	Type	Purpose
x-rcli:processes	ArrayOf(Process)	Processes to manage.
x-rcli:files	ArrayOf(File)	Stored uploaded files.
artifact	Artifact	File content for upload.
features	Features	Actuator capabilities.

Table 5.4: Targets Used in the RCLI Profile

5.3.3 Arguments

RCLI extends the base Arguments with an additional profile-specific parameter, `storage_file`, which specifies an optional save-to location.

Name	Type	Card.	Purpose
response_requested	ResponseType	0..1	Level of response.
start_time	Date-Time	0..1	When to start.
stop_time	Date-Time	0..1	When to stop.
duration	Duration	0..1	Run time limit.
storage	File	0..1	Save-to location.

Table 5.5: Arguments Used in the RCLI Profile

Argument Usage Matrix

Argument	start	stop	copy	query	delete
response_requested	X	X	X	X	X
start_time	X				
stop_time	X	X			
duration	X	X			
storage			X		

Table 5.6: Argument Applicability Across Actions

The table summarizes the applicability of each argument with respect to the actions supported by the RCLI profile. An “X” indicates that a specific argument can be

used in combination with the corresponding action, while empty cells denote that the argument is not relevant for that operation. For example, temporal parameters such as `start_time`, `stop_time`, and `duration` are meaningful only for the `start` and `stop` actions, whereas the `storage` argument is exclusively used with the `copy` action to specify the save location. The `response_requested` argument, on the other hand, is applicable to all actions.

5.3.4 Actuator Specifiers

At the current stage of implementation, the RCLI profile does not define any specific actuator specifiers. This design choice reflects the fact that all commands are directed to a single instance of the RCLI actuator running on the Consumer host.

5.4 Command Matrix

Target	Query	Start	Stop	Copy	Delete
features	X				
artifact				X	
x-rcli:processes	X	X	X		
x-rcli:files	X				X

Table 5.7: Allowed Action–Target Pairs in the RCLI Profile

This matrix describes the combinations of OpenC2 actions and targets that are permitted within the RCLI profile. Each row represents a possible target, while each column corresponds to an OpenC2 action. An “X” indicates that the specific action can be applied to the corresponding target, whereas empty cells denote unsupported combinations. For example, the `query` action can be used to retrieve information about available `features`, running `x-rcli:processes`, or managed `x-rcli:files`, while the `copy` action is applicable only to the `artifact` target. This table therefore provides a concise overview of the operational capabilities exposed by the RCLI actuator profile.

5.4.1 Features

Type: Features (ArrayOf(Feature))

The `features` target allows a Producer to query the Consumer for its OpenC2 capabilities and RCLI-specific extensions. It enables interoperability and discovery

of actuator functionality. The RCLI profile extends the feature list to include the allowlisted CLI commands (`clicommands`) that can be remotely executed. This mechanism provides Producers with secure visibility into supported commands and their argument patterns.

5.4.2 Artifact

Type: Artifact (Record)

The `artifact` target is used with the `copy` action to transfer files to the actuator host. Artifacts may include either binary payloads or URI references for remote retrieval. Once copied, the actuator records the file in its local database and associates it with the identity of the producer that performed the transfer.

5.4.3 x-rcli:processes

Type: ArrayOf(Process)

The `x-rcli:processes` target supports remote management of system processes, allowing Producers to start, stop, or query processes through standardized OpenC2 commands. Each `Process` entry may include attributes such as `name`, `pid`, `command_line`, or a reference to an `executable` file. The `start` action is used to launch an allowlisted command or executable on the actuator host, with optional arguments like `start_time`, `stop_time`, and `duration` that enable scheduled execution. Each started process is tracked by its PID and associated with the originating producer. The `stop` action terminates one or more processes previously started by the same producer, with authorization enforced through producer identifiers to prevent cross-tenant interference. The `query` action retrieves the current state of one or more processes, and if no PID is specified, all processes owned by the querying producer are returned.

5.4.4 x-rcli:files

Type: ArrayOf(File)

The `x-rcli:files` target supports querying and deleting files that were previously uploaded by a producer. Query actions return the list of accessible files and related metadata, while delete actions remove files from the actuator's local storage area. File isolation guarantees that producers can access or delete only their own uploaded files.

5.5 Response Components

This section defines the Response Components relevant to the RCLI Actuator Profile. An OpenC2 Response in this context conveys the outcome of an issued command, including its status code, descriptive text, and any associated results.

Type: OpenC2-Response (Map)

Name	Type	Card.	Description
<code>status</code>	Status-Code	1	The status code indicating the outcome of the command.
<code>status_text</code>	String	1	Human-readable description of the response status.
<code>results</code>	Results	0..1	The data returned by the actuator in response to the command.

Table 5.8: Fields Composing an OpenC2 Response in the RCLI Profile

5.5.1 Response Status Code

Type: Status-Code (Enumerated.ID)

The RCLI profile implements standard OpenC2 response codes with extensions relevant to process control and file management. Each code reflects the actuator's interpretation of the requested action outcome.

Code	Status	Description
102	Processing	Request accepted; execution pending or scheduled.
200	OK	Command executed successfully.
400	Bad Request	Invalid syntax or unsupported parameters.
401	Unauthorized	Producer not authorized for this operation.
403	Forbidden	Action denied by policy or security rules.
404	Not Found	Requested process or file not found.
500	Internal Error	Unexpected actuator or system failure.
501	Not Implemented	Action or target not supported by this profile.

Table 5.9: RCLI Response Status Codes and Their Descriptions

5.5.2 Common Results

These fields correspond to standard OpenC2 results, shared across all profiles, and included here for completeness.

Type: Results (Record 0..*)

ID	Name	Type	Card.	Description
1	versions	Version unique	0..*	List of OpenC2 language versions supported by this actuator.
2	profiles	ArrayOf(Nsid)	0..1	List of profiles supported by this actuator (e.g., <code>x-rcli</code>).
3	pairs	Action-Targets	0..1	List of valid target types per supported action.
4	rate_limit	Number	0..1	Maximum number of commands per minute supported by policy.

Table 5.10: Common OpenC2 Result Fields in the RCLI Context

5.5.3 Specific Results

The RCLI profile extends the standard OpenC2 results to report actuator-specific information such as process status, file operations, and available commands. These results are returned in the `results` object of an OpenC2 Response.

Type: Results (Map 0..*)

ID	Name	Type	Card.
2051	clicommands	Processes	0..1
2052	process_status	Processes	0..1
2053	file_status	Files	0..1

Table 5.11: RCLI Profile-Specific Result Fields

Usage Requirements

- `clicommands` — Returned only when `action=query`, `target=features`. This result lists all allowlisted command-line utilities and their corresponding parameters that can be executed through the actuator.
- `process_status` — Returned when the target is `x-rcli:processes` for `start`, `stop`, or `query` actions. It provides the current state of one or more processes, including PID, name, command line, and execution status.

- `file_status` — Returned for `copy`, `query`, or `delete` actions on `x-rcli:files`. It includes information about file location, name, and optional hashes for integrity verification.

5.6 Data Types

The RCLI Profile introduces two profile-specific data types that extend the OpenC2 language to support process state monitoring: `Process` (extended) and `State` (enumerated). These types allow the actuator to represent detailed process runtime information, including execution status derived from the host system.

5.6.1 Feature (Extended)

Type: Feature (Enumerated)

The `Feature` type defines the possible results returned from a `query features` command. In the RCLI profile, the base enumeration is extended to include the additional entry `clicommands`, which lists the allowlisted command-line instructions that can be executed by the actuator. This extension enables Producers to discover available CLI commands in a secure and standardized way.

ID	Name	Description
1	versions	List of OpenC2 language versions supported by this actuator.
2	profiles	List of profiles supported by this actuator.
3	pairs	List of supported actions and applicable targets.
4	rate_limit	Maximum number of commands per minute supported by design or policy.
5	clicommands	List of allowlisted command-line instructions that the actuator can execute remotely. This extension is specific to the RCLI profile.

Table 5.12: Extended Feature Enumeration Including the RCLI-Specific Entry `clicommands`

5.6.2 Process (Extended)

Type: Process (Record)

The `Process` type extends the standard OpenC2 `Process` definition by adding a new field, `state`, which encapsulates the runtime state of a process. This field provides visibility into the execution lifecycle of a process on the actuator system.

ID	Name	Type	Card.	Description
1	pid	Integer	0..1	Unique identifier assigned to the process by the operating system.
2	name	String	0..1	Name or label of the process. This may be temporary until a pid is assigned.
3	command_line	String	0..1	Command and arguments used to start the process.
4	executable	File	0..1	File object describing the executable used to create the process.
5	state	State	0..1	Current state of the process. The value is derived from the psutil status of the process and mapped to the State enumeration defined by the profile.

Table 5.13: Extended Process Data Type Definition

5.6.3 State

Type: State (Enumerated)

The **State** type is an enumerated data type that represents the possible runtime states of a process. These values are derived from system-level process information obtained through the Python **psutil** library and mapped into the OpenC2 profile namespace.

Value	Name	Description
1	running	The process is actively executing instructions.
2	sleeping	The process is waiting for an event to complete.
3	disk_sleep	The process is blocked in uninterruptible disk I/O.
4	stopped	The process has been stopped by a signal or user.
5	tracing_stop	The process is being traced or debugged.
6	zombie	The process is executed but still exists in the process table.
7	dead	The process is terminated and no longer active.
8	wake_kill	The process is being awakened to be killed.
9	waking	The process is in the process of waking up.
10	parked	The process is parked (kernel thread idle).
11	idle	The process is idle and waiting for work.
12	locked	The process is blocked, waiting for a resource lock.
13	waiting	The process is waiting for another event or resource.
14	suspended	The process is suspended by the system or user.
15	not_found	The process does not exist or has already terminated.

Table 5.14: State Enumerated Type Definition

5.7 Implementation Details and Security Considerations

This section documents implementation specifics of the RCLI actuator and highlights the security design decisions taken during development. File paths referenced below correspond to the implementation under `src/openc2lib/actuators/rcli`.

5.7.1 Overall structure and dispatching

The actuator is implemented around a central dispatcher class (`RCLIActuator` in `rcli_actuator.py`) that performs three sequential responsibilities for every incoming OpenC2 message: (1) profile and schema validation, (2) actuator addressing confirmation, and (3) deterministic dispatch to a dedicated action handler module. Validation is applied early to fail fast and to avoid unnecessary exposure of host-side utilities. Dispatch logic uses an explicit mapping from action enum values to handler modules to keep the control flow transparent and testable.

5.7.2 Input validation and profile enforcement

Validation is enforced in multiple layers:

- **Profile-level checks:** Before any action code executes, the actuator validates the action/target pair and argument types against the RCLI profile. This prevents unsupported combinations and ensures typed arguments (e.g., `DateTime`, `Duration`) pass basic schema checks.
- **Command/flags validation:** A central CLI validator (`cli/commands.py`) consults a JSON allowlist (`cli/available_commands.json`). The validator enforces command names, allowed flags and expands short flags to prevent bundled-flag bypasses. It also applies simple pattern checks to reject known-dangerous substrings.
- **Path and operand constraints:** For commands that accept path operands, the validator checks whether the command explicitly permits path tokens (via `allow_path` in the allowlist). This reduces implicit acceptance of arbitrary path expressions.

5.7.3 Command execution model and hardening notes

To reduce the attack surface, the implementation separates two execution branches: an *allowlisted shell command* branch and an *executable/artifact* branch. Important implementation notes and mitigations include:

- **Allowlist-first design:** Commands defined in the allowlist are the only commands constructed using validated token sets. The code constructs the execution string from validated tokens rather than running raw input.
- **Short-flag expansion:** The implementation expands bundled short flags into discrete flags prior to validation to prevent the classic short-flag bypass (e.g., `-abc` $\rightarrow$ `-a -b -c`).
- **Planned hardening (implemented partially):** Wherever possible the code attempts to avoid shell interpolation of unchecked strings. However, because the current code uses `subprocess.Popen(..., shell=True)` in `process_utils.open_process`, we explicitly flag this as a high-risk area and have documented the replacement with `shell=False` and positional argv building as the intended mitigation.
- **Executable branch controls:** Executable/artifact handling (docker-run wrappers in the implementation) is subject to stricter checks in code comments: an explicit allowlist, hash pinning, and signature verification are recommended. The present implementation emits warnings and records metadata when this branch is used.

5.7.4 Process lifecycle, authorization and persistence

Process management utilities consolidate process creation, monitoring and termination in `utils/process_utils.py`. Key implementation decisions:

- **Per-producer PID tracking:** Launched processes are persisted to an on-disk SQLite database (`database/SQLDB.py`) with a producer identifier (default: OS username) and the PID. This enables ownership checks for subsequent operations.
- **Authorization checks:** All operations that reference PIDs consult the database to enforce that only PIDs created by the same producer identity may be queried or terminated.
- **Graceful termination flow:** Termination logic kills process trees in a staged manner (children first, then parent) and updates the database atomically to avoid stale records. Timeouts and escalation to SIGKILL are implemented to ensure deterministic cleanup.
- **Audit metadata:** For each process lifecycle event the actuator records minimal structured metadata (producer, PID, command tokens, timestamp) to the DB/log to allow post-hoc audit while minimizing leakage of sensitive content.

5.7.5 File/artifact handling and integrity

File utilities are grouped under `utils/files_utils.py` and follow a write-and-register workflow:

- **Scoped storage root:** Artifacts are written under `/opt/<PRODUCER_ID>/...` to enforce a simple per-producer isolation boundary. The code composes paths using a controlled helper and (where possible) records the intended relative path in DB rather than trusting caller-provided absolute paths.
- **Payload acquisition:** The implementation accepts both direct binary payloads and remote URIs. Remote fetches use the `requests` library; the implementation documents the need for timeouts, size caps and scheme restrictions (`https`) and includes these as TODO-style safe-guards in the code.
- **Hashing and integrity:** For non-URI uploads the code computes and stores an MD5 digest together with file metadata and compares against caller-supplied hashes when provided. The code documents MD5 weaknesses and recommends SHA-256 adoption; pragmatically MD5 support is kept for backward compatibility while SHA-256 verification is recommended in hardening notes.
- **Binary-safe writes:** The code includes write-paths for binary payloads; earlier versions mistakenly used text-mode writes for bytes, which the implementation corrects to explicit binary file writes (`wb`) to avoid corruption.
- **Deletion authorization:** File deletion routines first consult the DB to ensure the file is owned by the requesting producer and only then remove both the filesystem entry and DB record.

5.7.6 Scheduling and timing model

Start/stop scheduling accepts both absolute `DateTime` and `Duration` arguments. Implementation details include:

- **Centralized argument handling:** A scheduling helper computes required sleep intervals and validates argument types before scheduling via `threading.Timer`. This keeps scheduling logic consistent across actions.
- **Timer-based execution:** Scheduled actions spawn lightweight timers that call the same handler functions used for immediate execution, ensuring identical authorization and logging paths. The approach is simple and reliable for low-volume scheduling but is documented as unsuitable for high-throughput production where persistent schedulers or job queues would be required.

5.7.7 Response normalization and logging

Responses are normalized to OpenC2 response objects by a dedicated handler (`handlers/response_handler.py`). Important choices:

- **Consistent status codes:** All action handlers return standardized OpenC2 status codes and structured result payloads, which simplifies orchestration and automated decision-making.
- **Structured logging:** The implementation logs structured events (producer, action, status) and limits free-form command text in high-verbosity logs to reduce leakage. The code also annotates logs when high-risk branches are taken (e.g., executable/artifact execution).
- **Redaction guidance:** Several logging calls are annotated with comments recommending redaction of sensitive arguments when logs are exported or sent to centralized telemetry.

5.7.8 Implemented mitigations and residual risks

During implementation several explicit mitigations were applied, while other items are documented for future hardening:

- **Applied mitigations:**
 - Allowlist-driven command acceptance with explicit flag validation.
 - Per-producer PID and file registration preventing cross-producer operations.
 - Database-backed authorization checks for both processes and files.
 - Staged termination logic and DB cleanup to avoid orphaned records.
 - Binary-safe file writes and stored integrity digests for uploaded artifacts.
- **Known residual risks and notes:**
 - The current execution uses `shell=True` in a small number of places; this is a documented high-risk area and the replacement with `shell=False` and `argv` arrays is recorded as an immediate remediation for future releases.
 - Remote URI downloads currently lack strict timeouts, size limits and domain allowlisting; these are annotated in the code and prioritized as mitigations against SSRF and resource exhaustion.
 - MD5 is used in some verification flows for backward compatibility; the implementation recommends and documents a migration plan to SHA-256-based verification and, when feasible, artifact signing.

- Path composition must be normalized and validated to avoid traversal or absolute-path overrides; the implementation contains helpers and comments enforcing a root-constraint model but also notes areas where stricter `realpath()` checks should be added.
- Producer identity is currently derived from local OS user configuration. For multi-tenant or remote deployments the code documents the necessity of deriving producer identity from the OpenC2 message originator or an authenticated channel binding.

Chapter 6

Network Flow Monitor

6.1 Objectives

The Network Flow Monitor (NFM) Actuator Profile defines a standardized OpenC2 interface for remotely controlling network flow monitoring functions. Its purpose is to unify the way Producers start, stop, and query monitoring activities across heterogeneous flow agents such as **nProbe** and **Packetbeat**, without requiring any awareness of the underlying implementation details.

In traditional deployments, each network probe exposes proprietary parameters and configuration models, complicating automation and integration. The NFM profile addresses this issue by providing a common abstraction layer that allows a Producer to initiate or terminate monitoring sessions, specify protocol or address filters, configure exporters and output formats, and discover supported capabilities of each monitoring actuator through a uniform syntax.

The primary goal of the NFM profile is therefore to decouple OpenC2 command semantics from the specific monitoring engine. Whether the Consumer runs **nProbe**, **Packetbeat**, or any other NetFlow-compliant tool, the same OpenC2 commands can be used to manage them consistently. Each actuator translates the common NFM syntax into its own configuration or command-line parameters, ensuring operational interoperability while maintaining a consistent command structure.

6.1.1 Operational Workflow

The Network Flow Monitor (NFM) profile enables a Producer to remotely manage network flow monitoring capabilities on a Consumer through standardized OpenC2 commands.

A typical interaction begins with feature discovery, where the Producer queries available interfaces, supported information elements, exporter options, and flow formats. Using this information, the Producer issues a start command containing a

FlowMonitor configuration that defines what traffic to monitor and how collected flows should be exported. The Consumer responds with a `monitor_id` that identifies the active monitoring session.

After a session is created, the Producer may query its status, retrieve configuration details, or stop monitoring when it is no longer required. Queries can include active filters, exporter settings, and interface statistics, enabling dynamic observation and tuning. Multiple monitoring sessions can run concurrently, each identified by a unique `monitor_id`. Producers can also adjust monitoring parameters or export configurations while a session is running to optimize data collection. Monitoring results can be aggregated or analyzed locally before being forwarded to collectors. The workflow is flexible, and only a subset of commands may be used depending on operational needs.

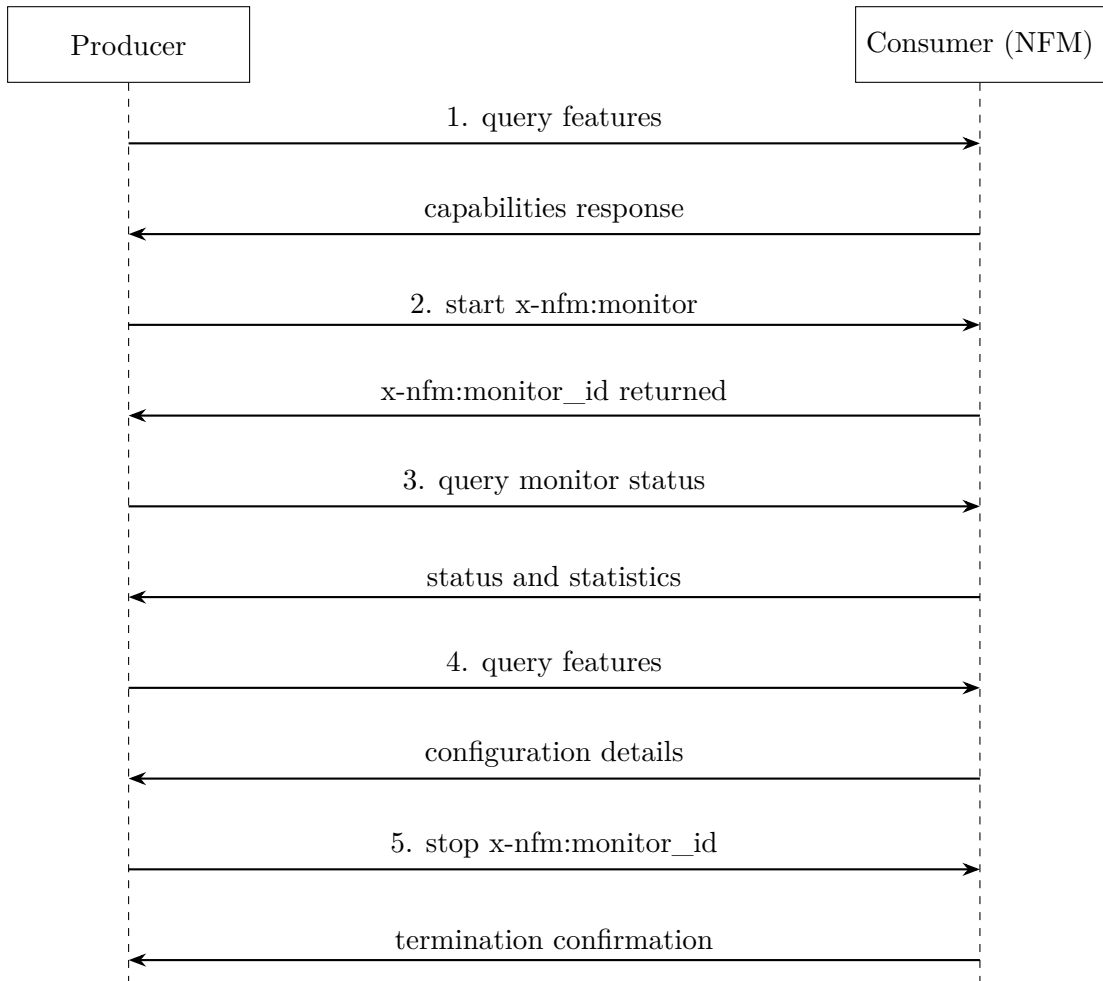


Figure 6.1: Typical operational workflow of the NFM actuator profile

Step	Action	Target	Purpose
1	query	features	Discover monitoring capabilities
2	start	x-nfm:monitor	Create monitoring session
3	query	x-nfm:monitor_id	Retrieve status of active monitor
4	query	features	Inspect configuration
5	stop	x-nfm:monitor_id	Terminate monitoring session

Table 6.1: Mapping of typical NFM workflow steps to OpenC2 commands

6.2 Data Model

The **NFM Profile** maintains a lightweight runtime state to manage the lifecycle of network flow monitoring sessions. This state records which monitoring processes are active, the producer that initiated them, and associated configuration details such as monitored interfaces and exporter parameters. The data is stored in a local SQLite database that acts as an operational registry, not a long-term audit log. It provides the actuator with context to track, identify, and control active monitors on a per-producer basis.

6.2.1 Database Structure

The database contains a single primary table, shown in Table 6.2, that tracks all running monitoring sessions. Each record is tied to the `uid` of the OpenC2 producer that initiated it, ensuring command ownership and preventing cross-producer interference or unauthorized termination.

Table	Fields
Monitors	uid, pid, monitor_id

Table 6.2: NFM Runtime Data Model (SQLite)

6.2.2 Monitor Tracking

The **Monitors** table records each active flow monitoring process spawned by the actuator in response to a `start` action. The `monitor_id` uniquely identifies a session, while `pid` links it to the local system process. The `uid` ensures that ownership is tied to the producer that created the monitor, providing fine-grained access control over monitoring sessions.

6.2.3 Session Lifecycle

When a producer issues a **start** command, a new record is added to the table. The process is launched asynchronously, and its **monitor_id** is returned to the producer in the *OpenC2 Results*. When a **stop** command is issued, the actuator looks up the associated entry, terminates the process, and removes the record. If scheduling arguments (**start_time**, **stop_time**, or **duration**) are used, the corresponding actions are deferred or timed accordingly.

6.2.4 Mapping to OpenC2 Responses

At runtime, *OpenC2 Results* objects are generated directly from the database contents. The **monitor_id** value is returned upon the successful creation of a monitoring session, providing a consistent reference to the active process. This mapping ensures uniform representation and synchronization of monitoring state across all NFM actuator implementations.

6.2.5 Security Enforcement

Each monitoring session is strictly bound to the **uid** of the producer that initiated it, guaranteeing the following security properties:

1. **Session isolation:** A producer can stop or inspect only the monitoring sessions it created.
2. **Process integrity:** System or unrelated monitoring processes are never exposed or modifiable through NFM.

These constraints preserve multi-tenant safety while maintaining full traceability of active flow monitoring sessions.

6.3 Command Components

This section describes the OpenC2 Command elements relevant to the NFM Profile: **Actions**, **Targets**, **Arguments**, and optional **Actuator Specifiers**. While the NFM Profile reuses Actions from the Language Specification, it introduces new Targets and profile-specific Arguments to control network flow monitoring behavior.

6.3.1 Actions

The NFM Profile uses the following Actions:

ID	Name	Description
3	query	Request capabilities or active monitor information.
9	start	Begin flow monitoring on specified interfaces.
10	stop	Terminate an active flow monitoring session.

Table 6.3: Actions Used in the NFM Profile

6.3.2 Targets

The NFM Profile defines two profile-specific Targets and reuses standard ones for feature discovery:

Name	Type	Purpose
x-nfm:monitor	FlowMonitor	Define configuration and filters for monitoring.
x-nfm:monitor_id	MonitorID	Reference to a running monitoring session.
features	Features	Actuator capabilities and supported options.

Table 6.4: Targets Used in the NFM Profile

6.3.3 Arguments

NFM extends the base Arguments with additional parameters specific to network flow export and scheduling:

Name	Type	Card.	Purpose
response_requested	ResponseType	0..1	Level of response.
start_time	Date-Time	0..1	When to start monitoring.
stop_time	Date-Time	0..1	When to stop monitoring.
duration	Duration	0..1	Run time limit.
exporter	Exporter	0..1	Where or how to export collected flows.
exporter_options	ExportOptions	0..1	Tuning options for exporter (e.g., format, sampling).

Table 6.5: Arguments Used in the NFM Profile

Argument Usage Matrix

Argument	query	start	stop
response_requested	X	X	X
start_time		X	
stop_time		X	X
duration		X	X
exporter		X	
exporter_options		X	

Table 6.6: Argument Applicability Across Actions

6.3.4 Actuator Specifiers

The NFM profile defines one actuator specifier used to select a specific probe instance.

Name	Type	Description
asset_id	String	Identifier of the target NFM actuator instance (e.g., <code>probe_x</code> , <code>probe_y</code>).

Table 6.7: Actuator Specifiers Used in the NFM Profile

If `asset_id` is omitted, the consumer dispatches the command to the default actuator for the NFM profile, i.e., the first registered NFM actuator in the local consumer configuration. In my setup, this default corresponds to the `NprobeActuator`.

6.4 Command Matrix

The command matrix shown in Table 6.8 illustrates the valid combinations of OpenC2 actions and targets defined by the NFM Profile. It describes how each command type (`query`, `start`, and `stop`) interacts with specific NFM targets to perform monitoring operations.

Target	Query	Start	Stop
features	X		
x-nfm:monitor		X	
x-nfm:monitor_id			X

Table 6.8: Allowed Action–Target Pairs in the NFM Profile

6.4.1 Features

Type: Features (ArrayOf(*Feature*))

The **features** target allows a Producer to query a Consumer for its OpenC2 capabilities and NFM-specific extensions. This discovery step enables interoperability and dynamic adjustment of commands to the Consumer’s supported monitoring agents, such as **nProbe** or **Packetbeat**. The NFM profile extends the **features** response to include supported monitoring options such as available network interfaces, exporters, flow formats, information elements, and filter capabilities. A Producer can issue a **query features** command to discover the Consumer’s supported OpenC2 capabilities. The query may include one or more specifiers such as versions, profiles, valid pairs, interfaces, information elements, exports, and export options.

6.4.2 x-nfm:monitor

Type: FlowMonitor (Record)

The **x-nfm:monitor** target defines a network flow monitoring configuration, specifying which interfaces to capture, which flow fields to export, and any filters to apply. A monitor represents an active or scheduled data collection session on a given probe or interface set. A Producer sends a **start** command to begin monitoring one or more interfaces. The configuration may include selected interfaces, information elements, protocol filters, and exporter settings. Optional scheduling arguments such as **start_time**, **stop_time**, or **duration** can be used to automate the execution time of the monitoring session.

6.4.3 x-nfm:monitor_id

Type: MonitorID (String)

The **x-nfm:monitor_id** target identifies an existing monitoring session that can be stopped or queried. This enables Producers to control active monitors by referencing their unique identifiers. A **stop** command terminates a running monitor, freeing resources and closing exporter outputs. The same target can also be queried to

obtain the list of currently active or scheduled monitors, including their identifiers, associated interfaces, and execution states. This mechanism provides a standardized way for Producers to manage, monitor, and synchronize active flow-monitoring sessions across different NFM actuator implementations.

6.5 Response Components

This section defines the Response Components relevant to the NFM Actuator Profile. An OpenC2 Response in this context conveys the outcome of a flow monitoring command, including its status code, descriptive text, and any associated results.

Type: OpenC2-Response (Map)

Name	Type	Card.	Description
<code>status</code>	Status-Code	1	The status code indicating the outcome of the monitoring command.
<code>status_text</code>	String	1	Human-readable description of the response status.
<code>results</code>	Results	0..1	Data returned by the actuator in response to the command.

Table 6.9: Fields Composing an OpenC2 Response in the NFM Profile

6.5.1 Response Status Code

Type: Status-Code (Enumerated.ID)

The NFM profile adopts standard OpenC2 response codes to indicate the outcome of monitoring operations, with minor extensions specific to network flow management. Each code reflects the actuator's interpretation of the command result, from successful execution to scheduling or error handling.

Code	Status	Description
102	Processing	Request accepted and scheduled (e.g., delayed start or stop).
200	OK	Command successfully executed (start, stop, or query).
400	Bad Request	Invalid or unsupported arguments in the command.
401	Unauthorized	Producer lacks permission for the requested action.
403	Forbidden	Action denied due to policy or safety restrictions.
404	Not Found	The specified monitor identifier was not found.
500	Internal Error	Unexpected actuator or exporter failure occurred.
501	Not Implemented	The command or argument is not supported.
503	Service Unavailable	Monitoring backend or exporter temporarily unreachable.

Table 6.10: NFM Response Status Codes and Descriptions

6.5.2 Common Results

These fields correspond to standard OpenC2 results, shared across all profiles, and included here for completeness.

Type: Results (Record 0..*)

ID	Name	Type	Card.	Description
1	versions	Version unique	0..*	List of OpenC2 language versions supported by this actuator.
2	profiles	ArrayOf(Nsid)	0..1	List of profiles supported by this actuator (e.g., x-nfm).
3	pairs	Action-Targets	0..1	List of valid target types per supported action.
4	rate_limit	Number	0..1	Maximum number of commands per minute supported by policy.

Table 6.11: Common OpenC2 Result Fields in the NFM Context

6.5.3 Specific Results

The NFM profile extends the standard OpenC2 results to report actuator-specific information such as monitoring status, exporter configuration, and discovered network capabilities. These results are returned in the **results** object of an OpenC2 Response.

Type: Results (Map 0..*)

ID	Name	Type	Card.
3051	interfaces	ArrayOf(Interface)	0..1
3052	information_elements	ArrayOf(IE)	0..1
3053	exports	ArrayOf(String)	0..1
3054	export_options	ArrayOf(String)	0..1
3055	flow_format	ArrayOf(String)	0..1
3056	monitor_id	MonitorID	0..1
3057	filters	ArrayOf(String)	0..1

Table 6.12: NFM Profile-Specific Result Fields

Usage Requirements

The NFM profile defines several profile-specific result fields used in OpenC2 responses. Their appearance depends on the executed command and target:

- **interfaces** — Returned when **action=query**, **target=features**, listing available or active network interfaces discovered on the actuator host.
- **information_elements** — Returned when querying supported flow attributes, such as source or destination IP, byte and packet counts, and other flow fields supported by the monitoring probe.
- **exports**, **export_options**, and **flow_format** — Returned as part of feature discovery or exporter configuration queries. These describe supported exporter types, tunable parameters such as sampling or buffer size, and available output formats (e.g., **NetFlow9**, **IPFIX**, **JSON**, or **CSV**).
- **monitor_id** — Returned upon successful execution of a **start** command to uniquely identify the created monitoring session.
- **filters** — Returned when querying the filter configuration, indicating the applied IPv4 or IPv6 connection filters currently active.

The NFM Profile introduces seven profile-specific data types that extend the OpenC2 language to support network flow monitoring: `FlowMonitor`, `MonitorID`, `Interface`, `Collector`, `ExporterOptions`, `FlowFormat`, and `IfaceType`. These data types define the schema for expressing monitoring configurations, tracking active sessions, and managing exporters and interfaces.

6.6 Data Types

6.6.1 Feature (Extended)

Type: Feature (Enumerated)

The `Feature` type defines the possible results returned from a `query features` command. In the NFM profile, this enumeration is extended to include additional entries that describe the capabilities of network flow monitoring actuators. These extensions allow Producers to query Consumers for available interfaces, supported information elements, exporter configurations, and filtering options — enabling interoperability and dynamic configuration across different monitoring agents such as `nProbe` and `Packetbeat`.

ID	Name	Description
1	versions	List of OpenC2 language versions supported by this actuator.
2	profiles	List of profiles supported by this actuator.
3	pairs	List of supported actions and applicable targets.
4	rate_limit	Maximum number of commands per minute supported by design or policy.
5	interfaces	Network interfaces available for flow monitoring.
6	information_elements	Supported flow fields such as source/destination IP, ports, bytes, and packets.
7	exports	Supported exporter types, e.g., file or collector.
8	export_options	Configurable exporter parameters such as sampling rate or output format.
9	flow_format	Supported output formats for exported flows (e.g., JSON, NetFlow, IPFIX).
10	filters	Available IPv4/IPv6 filters or protocol-level capture options.

Table 6.13: Extended Feature Enumeration for the NFM Profile

6.6.2 FlowMonitor

The **FlowMonitor** type defines the configuration of a monitoring session. It specifies the interfaces to capture traffic from, the flow fields to export, and any filters or exporter settings applied. This is the primary target for the **start** action in the NFM profile.

Type: FlowMonitor (Record)

ID	Name	Type	Card.	Description
1	interfaces	ArrayOf(Interface)	1..*	List of network interfaces to monitor for traffic flows.
2	information_elements	ArrayOf(IE)	0..*	Flow fields to collect (e.g., source IP, destination IP, bytes, packets).
3	filter_v4	ArrayOf(IPv4Connection)	0..*	Optional IPv4 filters to limit captured traffic.
4	filter_v6	ArrayOf(IPv6Connection)	0..*	Optional IPv6 filters to limit captured traffic.
5	exporter	Exporter	0..1	Exporter configuration defining where collected flows are sent or stored.
6	exporter_options	ExporterOptions	0..1	Advanced options for export behavior (e.g., sampling, aggregation, buffer size, format).

Table 6.14: FlowMonitor Data Type Definition

6.6.3 MonitorID

The **MonitorID** type represents a unique identifier assigned to each active flow monitoring session. It allows the Producer to reference and control running monitors (e.g., for stopping or querying them).

Type: MonitorID (String)

Name	Type	Description
monitor_id	String	Unique alphanumeric identifier returned by the actuator when a monitoring session is started. Used as a reference in subsequent stop or query commands.

Table 6.15: MonitorID Data Type Definition

6.6.4 Exporter

Type: Exporter (Record)

The **Exporter** type defines how collected flow data is exported from the monitoring actuator. It can specify a storage file, one or more collectors, or both. This flexible structure allows data to be written locally or forwarded to remote destinations such as flow collectors or analysis systems.

ID	Name	Type	Card.	Description
401	storage	File	0..1	Optional log file destination used for local data export.
402	collectors	ArrayOf(Collector)	0..1	List of downstream collectors to which flow data is transmitted.

Table 6.16: Exporter Data Type Definition

The **Exporter** must include at least one of the two fields: **storage** or **collectors**. Type validation ensures proper field types, and a helper method `get(key, default)` is provided for safe attribute access.

6.6.5 Collector

The **Collector** type defines the endpoint for flow data export. It specifies the remote collector's address and port, typically for NetFlow or IPFIX receivers.

Type: **Collector** (Record)

ID	Name	Type	Card.	Description
1	address	IPv4Net	1	Destination address of the collector in CIDR format.
2	port	Port	1	Network port used to receive exported flow records.

Table 6.17: Collector Data Type Definition

6.6.6 ExportOptions

The **ExportOptions** type defines additional tuning parameters for controlling exporter behavior, including sampling rate, aggregation, buffer size, and output format.

Type: **ExportOptions** (Record)

ID	Name	Type	Card.	Description
1	sampling	Integer	0..1	Sampling rate applied to packet or flow capture.
2	aggregate	Boolean	0..1	Indicates whether flow aggregation is enabled.
3	buffer	Integer	0..1	Buffer size (in records) before flushing exporter data.
4	timeout	Integer	0..1	Timeout (in seconds) for exporter flush or rotation.
5	format	FlowFormat	0..1	Format used for exporting flow records.

Table 6.18: ExportOptions Data Type Definition

6.6.7 IE

Type: IE (String)

The IE (Information Element) type represents a single flow field to be collected during monitoring, such as *source IP*, *destination IP*, *bytes*, or *packets*. It is implemented as a validated string subclass, ensuring that each information element is correctly defined and non-null at creation time.

ID	Name	Description
501	IE	String defining a specific flow attribute or metric collected by the actuator.

Table 6.19: Information Element (IE) Data Type Definition

6.6.8 Interface

The **Interface** type represents a network interface that can be monitored by the actuator. It captures the logical and physical attributes of a network interface, including addressing, status, and type.

Type: Interface (Record)

ID	Name	Type	Card.	Description
1	name	String	1	Interface name (e.g., <code>eth0</code> , <code>ens33</code>).
2	description	String	0..1	Optional textual description of the interface.
3	if_id	Integer	0..1	Unique system identifier for the interface.
4	ipv4_net	ArrayOf(IPv4Net)	0..*	List of IPv4 subnets assigned to the interface.
5	ipv6_net	ArrayOf(IPv6Net)	0..*	List of IPv6 subnets assigned to the interface.
6	mac_addr	MACAddr	0..1	MAC address of the network interface.
7	iface_type	IfaceType	0..1	Enumerated type describing the interface layer-2 category.
8	active	String	0..1	Indicates whether the interface is currently UP or DOWN.

Table 6.20: Interface Data Type Definition

6.6.9 IfaceType

The `IfaceType` type is an enumerated data type representing the underlying layer-2 type of a network interface. It helps identify the physical or virtual interface characteristics for monitoring and capability reporting.

Type: `IfaceType` (Enumerated)

Value	Name	Description
1	ethernet	Standard Ethernet interface.
2	loopback	Virtual loopback interface.
3	tunnel	Encapsulated tunnel interface (e.g., GRE, IPsec).
4	vlan	VLAN subinterface.
5	bridge	Network bridge interface.
6	wireless	Wi-Fi or wireless interface.
7	unknown	Undefined or vendor-specific interface type.

Table 6.21: IfaceType Enumerated Type Definition

6.6.10 FlowFormat

The `FlowFormat` type is an enumerated data type defining the supported export formats for flow data. This provides interoperability between different monitoring tools and collectors.

Type: `FlowFormat` (Enumerated)

Value	Name	Description
1	netflow5	Legacy NetFlow v5 export format.
2	netflow9	NetFlow v9 template-based export format.
3	ipfix	IETF-standard IP Flow Information Export format.
4	json	JSON-based format for file or API outputs.
5	csv	Comma-separated format for testing and analysis.

Table 6.22: FlowFormat Enumerated Type Definition

6.7 Implementation Details and Security Considerations

The Network Flow Monitor (NFM) actuator implements an OpenC2 profile that enables remote control over network flow monitoring on a host system. The actuator provides the ability to query monitoring capabilities and to start and stop network-flow capture sessions. The design supports multiple monitoring backends through a pluggable architecture, with two concrete implementations currently provided: an `NprobeActuator` for systems using *nProbe*, and a `PacketbeatActuator` for environments based on *Elastic Packetbeat*. Both implementations share a common orchestration layer, process lifecycle management utilities, structured capability discovery, and a consistent response model.

6.7.1 Architectural Overview

The central orchestration logic is defined in the `NetworkFlowMonitor` base class (`nfm_flow_monitor.py`). Incoming OpenC2 messages are first validated according to the NFM profile specification and then routed to one of three supported actions: `query`, `start`, or `stop`. The `query` action is implemented to return the monitoring capabilities of the host, including allowed interfaces, supported Information Elements (IEs), exporters, output formats, and filter options. These capability sets are dynamically derived from configuration files located under `configuration/`, which define available fields and probe-specific feature mappings.

The `start` and `stop` actions form the core operational lifecycle. The `start` action validates the supplied target and arguments, resolves configuration parameters such as interfaces, filtering expressions, sampling rates, and exporters, and then delegates execution to the backend-specific `_start_monitor` method. The `stop` action locates and terminates a running monitor using its `monitor_id`, ensuring correct cleanup of process state and persistent records.

6.7.2 Backend Implementations

nProbe Backend. The `NprobeActuator` constructs an `nProbe` command-line invocation from the target monitor specification and the supplied arguments. Interface selection is handled explicitly, and filter expressions are translated into BPF syntax through the `bpf_filter_translator.py` utility. Requested Information Elements are checked against the capability configuration, and unsupported requests result in a clear error response. Exporter options such as sampling, aggregation, buffer sizing, and timeout behavior are also applied here. The monitor process is then launched as a child process with a unique `monitor_id` assigned for later lookup.

Packetbeat Backend. The `PacketbeatActuator` instead generates a dedicated Packetbeat YAML configuration file per monitoring session. A base configuration template is loaded from an environment-defined location and modified to include the required network interfaces, BPF filters, sampling parameters, optional local output settings, and field selection rules. This configuration file is written to a monitor-specific directory, after which Packetbeat is launched with the `-c` option to reference the generated file. As in the `nProbe` implementation, the resulting monitor is registered using its `monitor_id` for subsequent management.

6.7.3 Process Management and Persistence

All running monitor processes are tracked using a local SQLite database (`nfmdata.db`), operated through the database interface defined in `SQLDB.py`. The database stores mappings between a `monitor_id`, its associated PID, and the identity of the OpenC2 producer initiating the request. This permits clean termination, prevents accidental interference between different users, and supports retrieval of state for `query` or `stop` operations. Process lifecycle management is handled through `process_utils.py`, which launches monitoring processes via `subprocess.Popen(..., shell=False)` and supervises their termination, escalating to forced kill if graceful shutdown fails.

6.7.4 Input Handling and Runtime Configuration

Environment variables control the paths to binaries, data directories, and interface whitelists. For example, `NPROBE_EXECUTABLE` and `PACKETBEAT_EXECUTABLE` specify executable paths, while `ALLOWED_INTERFACES` restricts the set of network interfaces available for monitoring. This separation enables deployment-time control over monitoring scope and host exposure. Capability and Information Element lookup is performed through the `ProbeConfigLoader`, which resolves per-asset configuration values from JSON maps keyed by `asset_id`.

6.7.5 Security Considerations

Several security-focused design choices were incorporated into the NFM actuator:

- **No shell interpolation.** All monitoring tools are launched using list-based `Popen` calls, preventing shell injection and disallowing interpretation of untrusted user input as command syntax.
- **Profile-enforced validation.** Action types, targets, and arguments are validated before execution, ensuring that only semantically valid commands reach the monitoring backends.
- **Interface restriction.** Network interfaces exposed for monitoring are explicitly whitelisted, reducing the risk of unauthorized packet capture.
- **Typed filter translation.** BPF expressions are generated from structured OpenC2 filter objects, rather than accepting raw filter strings, reducing injection and malformed filter risks.
- **Ownership tracking.** Monitor processes are scoped to the producing user identity and stored with a `PRODUCER_ID` identifier, preventing cross-user termination.
- **Controlled output logging.** Output streams are bounded and sanitized to prevent data leakage or log flooding.

6.7.6 Observed Risks and Future Hardening

During evaluation, several areas were identified for hardening. Environment-controlled executable paths and configuration directories should be validated or restricted to prevent binary hijacking or configuration injection. Monitor identifiers should be generated using cryptographically strong randomness to prevent guessing-based stop attacks. The SQLite database should be relocated to a protected directory with well-defined access control, and additional consistency checks should be added to defend against PID reuse. Further input range validation (e.g., sampling rates, buffer sizes) should be enforced to avoid resource exhaustion attacks.

Chapter 7

File Collection Log Monitoring

7.1 Objectives

The File Collection Log Monitoring (FCLM) Actuator Profile extends the OpenC2 language to provide a unified and interoperable framework for remote control of log collection activities. Its main objective is to standardize how log collection agents are configured and managed across heterogeneous platforms, promoting consistent orchestration of monitoring functions within complex cybersecurity architectures. Specifically, the FCLM profile defines a file monitoring framework for configuring the collection of logs, either locally or in a centralized location. It allows the coordination of multiple log sources—such as files, URIs, or sockets—through a single, technology-agnostic interface, ensuring that Producers can manage diverse collection processes in a uniform way.

By abstracting the underlying implementation details of different collectors, FCLM provides a common model for specifying what sources to monitor, how data should be gathered, and where collected logs should be exported. This approach enhances automation, scalability, and interoperability among security components by decoupling command semantics from the specific mechanisms of each tool.

7.1.1 Operational Workflow

The File Collection Log Monitoring (FCLM) profile allows a Producer to remotely manage log collection and export operations on a Consumer through standardized OpenC2 commands.

A typical workflow begins with feature discovery, where the Producer queries supported export fields, exporters, import controls, and available monitoring

sources. Next, the Producer issues a start command targeting `x-fclm:monitor` with configuration details such as file paths, sockets, exporter destinations, and optional timing arguments. The Consumer responds with a unique `x-fclm:monitor_id` identifying the active monitoring session.

Once monitoring is active, the Producer may query the session for status, collected fields, or exporter configurations. When finished, the Producer issues a stop command using the corresponding `x-fclm:monitor_id`. The workflow is flexible, and commands can be issued partially; for instance, a Producer may start monitoring without querying features or stop sessions without retrieving status.

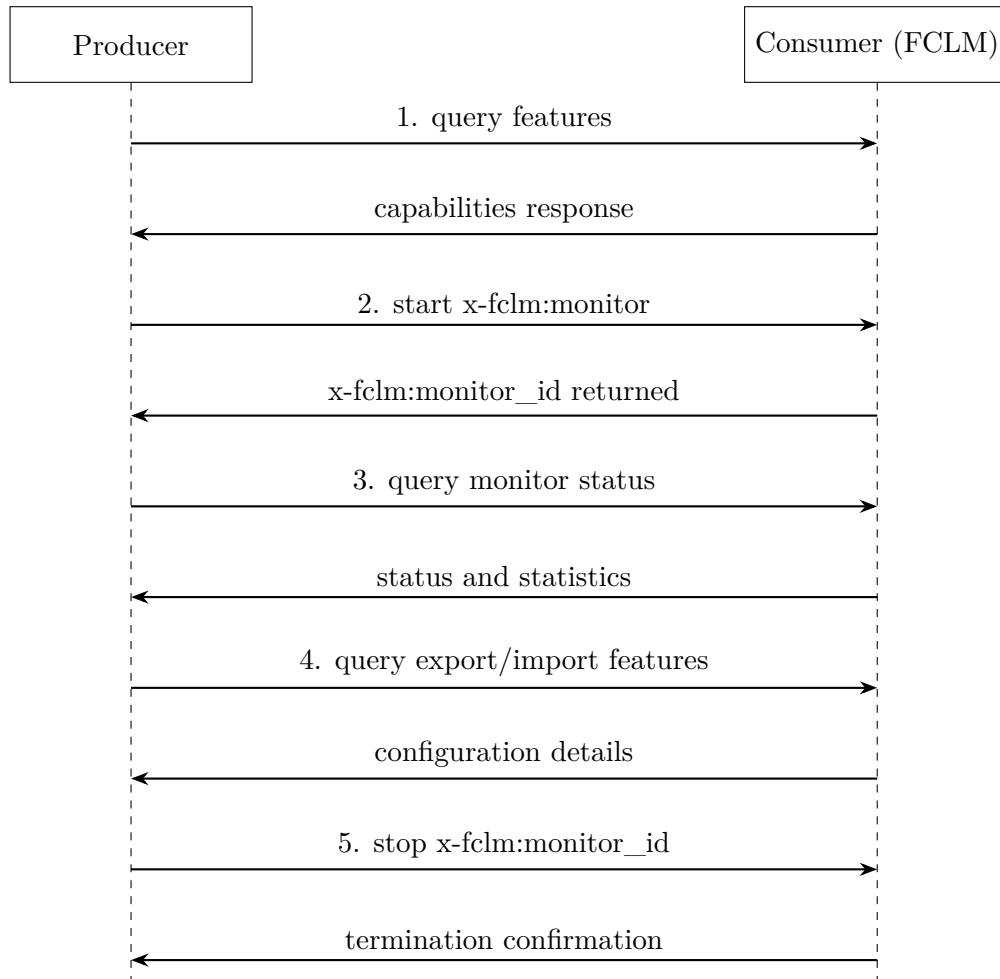


Figure 7.1: Typical operational workflow of the FCLM actuator profile

Table 7.1: Representative Command Flow for FCLM

Step	Action	Target
1	query	features
2	start	x-fclm:monitor
3	query	x-fclm:monitor
4	stop	x-fclm:monitor_id

7.2 Data Model

The **FCLM Profile** maintains a lightweight runtime database to track active monitoring processes and their associated metadata. This runtime state allows the actuator to map each running Filebeat instance to the OpenC2 producer that initiated it, ensuring ownership isolation and secure lifecycle management. The implementation relies on a local SQLite database colocated with the actuator. The database serves as a transient state store for process identifiers (PIDs) and monitoring session references, rather than a persistent audit repository.

7.2.1 Database Structure

The database schema consists of a single table named **pids**, as summarized in Table 7.2. This table associates each running process with the corresponding OpenC2 producer `uid` and the logical `monitor_id` used within FCLM commands. The unique constraint on the `pid` field prevents duplicate tracking entries and ensures consistent state management when multiple monitoring sessions are active.

Table	Fields
pids	uid, pid, monitor_id

Table 7.2: FCLM Runtime Data Model (SQLite)

7.2.2 Monitor Tracking

Each record in the **pids** table represents an active log monitoring process started by an OpenC2 `start` command. The `uid` identifies the producer that issued the command, the `pid` stores the process identifier of the Filebeat instance responsible for collection, and the optional `monitor_id` uniquely links the process to the OpenC2 command context. This mapping allows the actuator to locate and control individual monitoring sessions, such as terminating them via the `stop` command or reporting their status upon query.

7.2.3 Lifecycle Management

The actuator inserts a new record into the database whenever a monitoring process is successfully launched and removes the record when the process terminates or is explicitly stopped. Duplicate insertions are ignored through SQLite's **UNIQUE** constraint on the `pid` field. This design keeps the runtime database lightweight, stateless across restarts, and easy to reconstruct from the current process state.

7.2.4 Security Enforcement

By linking every process identifier to the originating producer's `uid`, the actuator enforces strict session ownership and prevents unauthorized interference between concurrent monitoring operations. This mechanism ensures that:

1. Only the producer that initiated a monitoring session can stop or query it.
2. Monitoring metadata remains isolated, preserving confidentiality in multi-tenant environments.

Such isolation and ownership guarantees align with the security principles of the OpenC2 framework, ensuring trustworthy and controlled execution of log collection operations.

7.3 Command Components

This section describes the OpenC2 Command elements relevant to the FCLM Profile: **Actions**, **Targets**, **Arguments**, and optional **Actuator Specifiers**. The FCLM Profile reuses standard OpenC2 Actions while introducing new Targets and Argument extensions for log collection and monitoring control.

7.3.1 Actions

The FCLM Profile supports a minimal, well-defined set of Actions used to manage monitoring sessions and query available capabilities.

ID	Name	Description
3	query	Retrieve available monitoring capabilities and configurations.
9	start	Initiate a new log monitoring session.
10	stop	Terminate an active monitoring session.

Table 7.3: Actions Used in the FCLM Profile

7.3.2 Targets

The FCLM Profile introduces new targets specific to log monitoring and extends the OpenC2 model to describe monitored sources and session identifiers.

Name	Type	Purpose
x-fclm:monitor	LogMonitor	Define source to monitor (file, URI, or socket).
x-fclm:monitor_id	MonitorID	Identify a running monitor.
features	Features	Report actuator capabilities and FCLM extensions.

Table 7.4: Targets Used in the FCLM Profile

7.3.3 Arguments

FCLM extends the base OpenC2 Arguments with additional profile-specific parameters to configure exporters, import controls, and export field selections.

Name	Type	Card.	Purpose
response_requested	ResponseType	0..1	Level of response detail.
start_time	Date-Time	0..1	When to begin monitoring.
stop_time	Date-Time	0..1	When to stop monitoring.
duration	Duration	0..1	How long monitoring should run.
log_exporter	Exporter	0..1	Defines where and how logs are exported.
export_fields	ArrayOf(EF)	0..1	Fields included in exported logs.
import_controls	ImportOptions	0..1	Tune ingestion parameters (scan rate, backoff, etc.).

Table 7.5: Arguments Used in the FCLM Profile

Argument Usage Matrix

The following matrix illustrates which Arguments are applicable to each Action within the FCLM Profile.

Argument	query	start	stop
response_requested	X	X	X
start_time		X	
stop_time		X	X
duration		X	X
log_exporter		X	
export_fields		X	
import_controls		X	

Table 7.6: Argument Applicability Across FCLM Actions

7.3.4 Actuator Specifiers

The FCLM profile defines one actuator specifier to address a specific log-collection actuator instance (e.g., an agent running on a host or container).

Name	Type	Description
<code>asset_id</code>	String	Identifier of the target FCLM actuator instance (e.g., <code>agent_x</code>).

Table 7.7: Actuator Specifiers Used in the FCLM Profile

If `asset_id` is omitted, the consumer dispatches the command to the default FCLM actuator selected by local registration order. In my setup, this corresponds to the Filebeat-based actuator instance.

7.4 Command Matrix

Table 7.8 summarizes the allowed combinations of actions and targets supported by the File Collection Log Monitoring (FCLM) profile. The profile defines a minimal yet functional set of commands sufficient for managing the lifecycle of log monitoring sessions.

Target	Query	Start	Stop
<code>features</code>	X		
<code>x-fclm:monitor</code>		X	
<code>x-fclm:monitor_id</code>			X

Table 7.8: Allowed Action–Target Pairs in the FCLM Profile

The `query` action enables a Producer to request information about the Consumer’s supported capabilities, configuration options, and runtime features. The `start` action initiates a new log monitoring session for a specified source, while the `stop` action terminates an active monitor identified by its unique `monitor_id`. This design ensures a clear and intuitive command model that mirrors the operational lifecycle of monitoring activities.

7.4.1 Features

The `features` target allows a Producer to query the Consumer for its OpenC2 capabilities and FCLM-specific extensions. This mechanism supports dynamic

discovery and interoperability between different OpenC2 implementations. The returned information typically includes the supported OpenC2 versions, the list of active profiles, the permitted action–target pairs, and the FCLM-specific capabilities such as available exporters, import controls, and exportable fields.

Through this feature query, a Producer can adapt its commands dynamically to match the Consumer’s capabilities. For instance, it can discover which log sources can be monitored, which output formats and collectors are supported, and which import parameters can be tuned (such as scan frequency or backoff timing). This ensures flexibility and compatibility when coordinating operations across heterogeneous systems.

7.4.2 x-fclm:monitor

The `x-fclm:monitor` target represents the log source to be monitored. Depending on the configuration, the monitored source may be a local file, a remote URI, or a network socket. The target uses a *Choice* type, ensuring that each `start` command initiates the monitoring of a single source at a time. This approach simplifies management and allows the actuator to maintain independent monitoring processes for each input.

When a `start` command is issued, the Producer specifies the source type (for example, a file path or socket address) along with optional arguments defining exporters, import controls, and output fields. These parameters determine how the logs are read, filtered, and exported to collectors. Upon successful initiation, the Consumer returns a confirmation message including a unique `monitor_id`, which serves as a reference for future control operations such as stopping or querying the session.

7.4.3 x-fclm:monitor_id

The `x-fclm:monitor_id` target identifies an active monitoring session that was previously started through a `start` command. This identifier is required when issuing a `stop` command to terminate a specific monitoring process. The use of `monitor_id` ensures that each stop operation explicitly targets the correct session, maintaining strict ownership and avoiding accidental termination of unrelated monitors.

FCLM also supports scheduled termination, allowing a Producer to specify timing parameters such as `stop_time` or `duration`. These arguments enable delayed or time-bound execution, making it possible to automate the start and stop lifecycle of monitoring processes without manual intervention. This feature is particularly useful in long-running environments where periodic data collection is required.

7.5 Response Components

This section defines the Response Components relevant to the FCLM Actuator Profile. An OpenC2 Response in this context conveys the outcome of a log monitoring command, including its status code, descriptive text, and any resulting data such as active monitor identifiers or supported configuration options.

Type: OpenC2-Response (Map)

Name	Type	Card.	Description
<code>status</code>	Status-Code	1	Indicates the result of the command execution.
<code>status_text</code>	String	1	Human-readable explanation of the returned status.
<code>results</code>	Results	0..1	Data structure returned by the actuator in response to a command.

Table 7.9: Fields Composing an OpenC2 Response in the FCLM Profile

7.5.1 Response Status Code

Type: Status-Code (Enumerated.ID)

The FCLM profile implements standard OpenC2 response codes while adding context specific to log monitoring and export management. Each code reflects the actuator’s evaluation of the requested action outcome.

Response Codes and Their Meaning

Code	Status	Description
102	Processing	Request accepted and running asynchronously.
200	OK	Command executed successfully (e.g., monitor started, stopped, or queried).
400	Bad Request	Invalid or missing parameters in the command.
401	Unauthorized	Producer lacks permission for the requested action.
403	Forbidden	Operation blocked by policy or security rules.
404	Not Found	Target monitor_id does not exist or is inactive.
500	Internal Error	Unexpected failure in actuator or backend.
501	Not Implemented	Action or target not supported by FCLM.
503	Service Unavailable	Monitoring subsystem temporarily overloaded.

Table 7.10: FCLM Response Status Codes

7.5.2 Common Results

These fields represent standard OpenC2 result properties, shared across all profiles, and included here for reference.

Type: Results (Record 0..*)

ID	Name	Type	Card.	Description
1	versions	Version	0..*	Supported OpenC2 language versions.
2	profiles	ArrayOf(Nsid)	0..1	List of active profiles implemented by this actuator (e.g., <code>x-fclm</code>).
3	pairs	Action-Targets	0..1	Mapping of valid actions to corresponding targets.
4	rate_limit	Number	0..1	Maximum commands per minute as defined by policy.

Table 7.11: Common OpenC2 Result Fields in the FCLM Context

7.5.3 Specific Results

The FCLM profile extends the OpenC2 `results` object to return monitoring- and export-related data. These results provide visibility into running sessions, configuration states, and available monitoring features.

Type: Results (Map 0..*)

ID	Name	Type	Card.
3051	monitor_id	String	0..1
3052	export_fields	ArrayOf(EF)	0..1
3053	exports_config	Array	0..1
3054	imports_config	Array	0..1
3055	import_controls	ImportOptions	0..1

Table 7.12: FCLM Profile-Specific Result Fields

Usage Requirements

- `monitor_id` — Returned after a successful `start` action to uniquely identify the running monitoring session. It is later used with `stop` or `query` to manage or inspect that session.
- `export_fields` — Returned when querying `features` to indicate which log fields can be exported by the actuator.

- **exports_config** — Returned with querying **features** to summarize the exporter configuration (e.g., output destination, format, or buffering parameters).
- **imports_config** — Returned when reporting which ingestion or preprocessing components are supported by the actuator.
- **import_controls** — Returned to indicate adjustable ingestion parameters, such as input scan rate, retry/backoff behavior, or resource usage limits.

7.6 Data Types

The FCLM Profile introduces several profile-specific data types that extend the OpenC2 language to describe log monitoring, exporting, and import control configurations. These data types define how monitored sources, export destinations, and ingestion parameters are represented and managed within FCLM commands and responses.

7.6.1 Feature (Extended)

Type: Feature (Enumerated)

The **Feature** type indicates the capability set returned in response to a **query features** command. The FCLM profile extends this enumeration to report log export attributes and ingestion control options specific to file- and socket- based log monitoring. These extensions allow Producers to discover the supported exporter types, configurable export fields, and available import controls, thus enabling dynamic setup and management of monitoring sessions.

ID	Name	Description
1	versions	Supported OpenC2 language versions.
2	profiles	Supported OpenC2 profiles.
3	pairs	Allowed action/target combinations.
4	rate_limit	Command rate limits enforced by the actuator.
11	export_fields	Supported fields that may be included in exported logs.
12	exports_config	Available exporter configurations (e.g., format, output path, endpoint).
13	imports_config	Supported ingestion or preprocessing configurations.
14	import_controls	Tunable import parameters (e.g., scan rate, retry/backoff thresholds).

Table 7.13: Extended Feature Enumeration for the FCLM Profile

7.6.2 Exporter

The **Exporter** type specifies where and how monitored log data should be exported. It allows defining both local storage and remote collectors to which logs are forwarded for further processing or aggregation.

Type: Exporter (Record)

ID	Name	Type	Card.	Description
1	storage	File	0..1	File object indicating the path where logs are stored locally.
2	collectors	ArrayOf(Collector)	0..1	List of downstream log collectors or export destinations. Each collector defines an address, port, and format.

Table 7.14: Exporter Data Type Definition

7.6.3 Collector

The **Collector** type defines an external system that receives log exports. Collectors may represent SIEM systems, log servers, or network endpoints that consume monitoring data in a structured format.

Type: Collector (Record)

ID	Name	Type	Card.	Description
1	address	IPv4Net	1	IP address or subnet of the collector host.
2	port	Integer	0..1	Port number on which the collector listens.
3	format	FileFormat	0..1	Data encoding format for export (e.g., JSON, CSV, YAML).

Table 7.15: Collector Data Type Definition

7.6.4 FileFormat

The `FileFormat` type enumerates the supported data output formats for exported log files.

Type: `FileFormat` (Enumerated)

Value	Name	Description
1	text	Plain-text log output.
2	json	JSON-encoded structured log output.
3	yaml	YAML-formatted log export.
4	csv	Comma-separated values (CSV) representation.
5	xml	XML-encoded export format.

Table 7.16: FileFormat Enumerated Type Definition

7.6.5 ImportOptions

The `ImportOptions` type defines tuning parameters for log ingestion, such as scanning intervals and backoff strategies. These options control how frequently monitored files or sockets are checked for new data and when inactive inputs are closed.

Type: `ImportOptions` (Map)

Name	Type	Description
<code>scan_frequency</code>	Duration	Interval between consecutive file scans.
<code>clean_inactive</code>	Duration	Time after which inactive sources are removed.
<code>max_backoff</code>	Duration	Maximum delay applied during exponential backoff.
<code>close_inactive</code>	Duration	Time after which inactive log streams are closed.
<code>ignore_older</code>	Duration	Ignores files older than this threshold.

Table 7.17: ImportOptions Map Type Definition

7.6.6 EF (Export Field)

The **EF** type is a string wrapper that identifies individual fields exported from monitored logs. These fields determine which data attributes (e.g., timestamps, metadata, payloads) are included in the exported record.

Type: EF (String)

Value	Name	Description
–	timestamp	The time each log entry was recorded.
–	metadata	Metadata about the monitored source or event.
–	payload	The raw log content extracted from the file or socket.

Table 7.18: EF String Wrapper Definition

7.6.7 Socket

The **Socket** type defines a network source for real-time log collection. It specifies connection details such as host address, port, and transport protocol.

Type: Socket (Record)

ID	Name	Type	Card.	Description
1	host	IPv4Net	1	IP address or network source of the log stream.
2	port	Integer	0..1	Network port number to collect logs from.
3	protocol	L4Protocol	0..1	Transport protocol used for collection (e.g., TCP, UDP).

Table 7.19: Socket Record Type Definition

7.7 Implementation Details and Security Considerations

The File Collection and Log Monitoring (FCLM) actuator implements an OpenC2 profile dedicated to controlling log collection activities across files, URIs, and sockets. Its main objective is to allow remote orchestration of collection agents such as Filebeat through standardized OpenC2 commands, while ensuring secure and traceable execution.

7.7.1 Implementation Overview

So far, only one actuator has been implemented. The FCLM actuator is composed of modular components that handle command processing, subprocess orchestration, configuration generation, and response management. The implementation resides under `src/openc2lib/actuators/fclm/`, with the following core modules:

- **FilebeatActuator** builds per-monitor YAML configurations and launches Filebeat instances (`fclm_actuator_filebeat.py`, line 19).
- **LogCollectionMonitor** defines the base class for lifecycle and command handling (`fclm_file_monitor.py`, line 23).
- **Process management utilities** handle subprocess creation, termination, and PID tracking in SQLite (`utils/process_utils.py`, lines 19–62) with the database managed in `database/SQLDB.py`.
- **Configuration loaders** process capability definitions and export field mappings from JSON files (`configuration/log_config_loader.py`, line 1).
- **Response and argument handlers** manage standardized replies and scheduled actions (`handlers/response_handler.py` and `handlers/argument_handler.py`).

Environment parameters, such as binary paths and configuration directories, are stored in `.env` to simplify deployment and runtime customization.

7.7.2 Execution Flow

The actuator processes OpenC2 commands through the `run()` function, which validates incoming messages and dispatches them to the proper action handlers: `query`, `start`, or `stop`.

Query. The query operation reports actuator capabilities, supported input/output formats, and FCLM-specific import and export options.

Start. The start handler validates arguments such as `response_requested`, `start_time`, `stop_time`, and `duration`. The `_start_monitor()` function then parses the target, constructs a Filebeat YAML configuration, and spawns the monitoring process using `subprocess.Popen(shell=False)`. Timers may be used for delayed starts and automatic stops.

Stop. Stop requests can target monitors by their logical identifier, which is resolved to the corresponding PID via the SQLite registry. The `terminate_process_and_children()` function ensures safe termination of the process tree and removal of PID entries.

7.7.3 Configuration and Process Control

Each monitor instance generates a dedicated Filebeat configuration under `$FILEBEAT_CONFIG_DIR`, derived from a base template specified in `.env`. The builder merges input parameters (filestream, HTTP JSON, or socket), export processors, and output sink definitions.

The database layer provides minimal schema management and parameterized queries to record running monitors, ensuring consistency between monitor identifiers and system processes. Output streams are partially captured to detect early failures while preventing log flooding and memory pressure.

7.7.4 Security Considerations

Security was a primary concern during the actuator's design and implementation. The following principles were applied to reduce risk and improve resilience.

Strengths

- **Safe process execution:** All subprocesses are invoked with `shell=False` to prevent shell injection vulnerabilities.
- **Strict typing and validation:** OpenC2 target fields use typed objects (URI, Socket, Duration), and invalid data is rejected early.
- **Controlled export mappings:** Export fields are resolved using pre-defined mappings. Unknown fields return `None` and are ignored.
- **Memory and log control:** Output capture is limited to a few lines, preventing log amplification or resource exhaustion.

- **Parameterized SQL:** SQLite statements are parameterized to mitigate injection risks.

Risks and Mitigations

Path Handling. File paths provided by users are directly used in YAML configurations. To prevent path traversal or overwriting sensitive files, all paths must be validated and restricted to approved directories before writing.

Environment Configuration. Environment variables define binary locations and directories. These should be immutable in production and not modifiable at runtime to prevent privilege escalation or redirection attacks.

Resource Usage. Concurrent monitors could exhaust disk space or system resources. Implementing quotas and limiting active monitors per user reduces the risk of resource depletion.

Access Control. Monitor ownership is bound to the producer identity. However, OpenC2-level authentication and authorization are enforced externally. Secure transport with mutual TLS and role-based authorization is recommended.

PID Resolution. Stopping processes by PID instead of logical ID may risk terminating unrelated processes. The function should validate that the PID belongs to a known monitor.

File Permissions. Configuration, log, and data directories should be accessible only by the dedicated service account. Running Filebeat as a non-root user is strongly recommended.

7.7.5 Cross-Cutting Security Controls

The following security mechanisms must be applied across all implemented actuators (FCLM, NFM, RCLI):

- **Authentication and Authorization:** The OpenC2 transport layer must enforce strong identity verification and access policies before commands reach actuators.
- **Least Privilege:** Filebeat and related agents should run under service accounts with minimal permissions, isolated from system and user data.

- **Input Validation:** Reject relative or invalid paths, normalize directories, and enforce safe parameter ranges for durations, hostnames, and file paths.
- **Process Accounting:** Track monitor ownership, creation time, and state to support auditability and prevent unauthorized terminations.
- **Secure Defaults:** Deny startup if critical configuration values are missing or paths fall outside of approved roots. YAML and JSON files must be validated before use.

Chapter 8

Validation and Testing

8.1 Overview

This section describes the validation methodology adopted to ensure the correctness and robustness of the developed OpenC2 profiles — RCLI, NFM, and FCLM. Validation was performed at two complementary levels:

1. **Data Type Validation:** verifying the internal `openc2lib` type system, including construction, field enforcement, and array and enumeration behavior.
2. **JSON Command and Response Validation:** verifying that profile commands and responses can be correctly encoded, decoded, transmitted via HTTP, and validated against their respective JSON Schemas.

The validation framework is entirely automated through `pytest` and YAML/JSON driven tests. Each test is categorized into “good” (expected to succeed) and “bad” (expected to fail) cases, ensuring both positive and negative coverage. All tests are available in the project Git repository. All validation tests were executed using `pytest` from the project root:

```
python -m pytest -q validation/openc2lib/test_types/{Profile}
python -m pytest -q validation/openc2lib/test_json/{Profile}
```

8.2 Data Type Validation

The `test_types` suite validates the correctness of the `openc2lib` type system for both core and profile-specific data. It verifies object construction from Python dictionaries, field type enforcement, enumeration mapping, `ArrayOf` container behavior, and that invalid inputs raise the appropriate exceptions. Most tests are

YAML-driven using the `parametrize_from_file` framework and are divided into “good” cases that must construct/validate successfully and “bad” cases that must fail with `TypeError` or `ValueError`.

8.2.1 RCLI Profile

Scope

RCLI validation covers the extended `Process` type, the `State` enumeration, arrays of processes and files, and `ArrayOf` behavior.

Tests

- **Process:** verifies construction from YAML, correct mapping of `state` strings to the `State` enumeration, and handling of missing or malformed fields.
- **State:** validates that valid names map to correct enum values and invalid names raise `KeyError`.
- **Processes (ArrayOf):** ensures element type checking, enforcement of maximum size, and acceptance of empty arrays.
- **Files and ArrayOf(File):** validates file object creation, correct hash field types (`md5`, `sha1`, `sha256`), and rejects malformed hashes or over-length arrays.

Results

Type	Good Tests	Good Passed	Bad Tests	Bad Passed
files	6	5	10	9
process	16	16	17	17
processes	5	5	11	11
state	15	15	6	6
Total	42	41	44	43

Table 8.1: RCLI Data Type Validation Summary

8.2.2 NFM Profile

Scope

The NFM validation ensures the correctness of collectors, exporters, interfaces, flow monitors, and associated enumerations such as `FlowFormat` and `IfaceType`.

Tests

- **Collector:** validates IPv4Net address and port correctness; malformed IPs or out-of-range ports must fail.
- **Exporter:** ensures at least one of **storage** or **collectors** is set and correctly typed.
- **ExportOptions:** checks integer fields for sampling and buffer sizes and that format values map to **FlowFormat**.
- **Interface:** verifies proper mapping of interface attributes, valid network notation, and array types.
- **FlowMonitor** / **MonitorID:** validates target behavior and element conversions, enforcing target semantics.

Results

Type	Good Tests	Good Passed	Bad Tests	Bad Passed
collector	26	26	16	16
export_options	7	7	9	9
exporter	6	6	7	7
flow_format	8	8	0	0
flow_monitor_target	6	6	7	7
ie	18	18	8	8
iface_type	11	11	10	10
interface	12	12	9	9
monitor_id_target	15	15	8	8
Total	109	109	74	74

Table 8.2: NFM Data Type Validation Summary

8.2.3 FCLM Profile

Scope

FCLM validation includes collector and exporter definitions, import options, log monitor targets, and socket and file format types.

Tests

- **Collector** / **Exporter:** ensures valid combinations of storage and collectors with correct address and port formats.

- **ImportOptions:** validates integer and duration fields (`scan_frequency`, `max_backoff`, etc.).
- **LogMonitor (Choice):** verifies that only one monitor variant (File, URI, or Socket) is defined and correctly shaped.
- **Socket:** tests host, port, and protocol validation for correct L4Protocol enumeration.
- **FileFormat and EF:** ensure correct mapping and error handling for invalid names or non-string values.

Results

Type	Good Tests	Good Passed	Bad Tests	Bad Passed
collector	10	10	15	15
ef	18	18	7	7
exporter	8	8	9	9
file_format	5	5	10	10
import_options	12	12	18	18
log_monitor_target	13	13	13	13
monitor_id_target	14	14	7	7
socket	12	12	16	16
Total	92	92	95	95

Table 8.3: FCLM Data Type Validation Summary

8.2.4 Findings and Resolution

During the `test_types` campaign, I identified two residual issues in the RCLI area that originate from the core `openc2lib` type system rather than from my profile implementation. The first concerns hash-length handling (e.g., MD5 validation behavior), and the second concerns permissive acceptance of nullable hash fields (e.g., `sha1: null`) in a path that should be stricter. I classify these as *non-critical* for my thesis scope because they are edge-case validation inconsistencies in shared core types, and they do not block the main RCLI profile modeling, command construction, or JSON command-level validation workflow that I implemented.

In contrast, `test_types` was very effective in revealing profile-level issues in my own implementations (NFM/FCLM), including datatype coercion corner cases, enum negative-test expectations, and some fixture misclassification between valid and invalid inputs. I resolved these by tightening profile-side validation logic, aligning tests with actual runtime behavior where appropriate, and correcting invalid test

fixtures. After these fixes, the NFM and FCLM type suites pass completely, which confirms that the profile-specific datatypes and validations I implemented are now consistent and robust.

8.3 JSON Command and Response Validation

The `test_json` suite validates end-to-end interoperability of OpenC2 JSON commands and responses for each profile. It confirms that commands can be decoded into `openc2lib.Command` objects, re-encoded to canonical JSON, and exchanged over HTTP in compliance with the profile JSON Schema.

8.3.1 RCLI Profile

Overview

The RCLI tests validate encoding/decoding and HTTP schema conformance for actions such as `query`, `start`, `stop`, `copy`, and `delete`. Good vectors reside under `validation/openc2-json-schema/tests/commands/good/rcli`, while bad vectors are located in the corresponding `/bad/rcli` directory.

Good Tests

- `test1_good.json`: query features (`clicommands`) with `x-rcli` args/actuator.
- `test2_good.json`: start processes (two entries) via `x-rcli:processes` with name and `command_line`.
- `test3_good.json`: stop processes by PID via `x-rcli:processes`.
- `test4_good.json`: copy artifact from URL to storage path/name using `x-rcli` args.
- `test5_good.json`: delete files via `x-rcli:files`.
- `test6_good.json`: start single process with executable and `command_line`, including `start_time`.
- `test7_good.json`: stop multiple processes by PID.
- `test8_good.json`: delete multiple files.
- `test9_good.json`: copy artifact (octet-stream) to storage path/name.
- `test10_good.json`: query standard features (versions, profiles, pairs).
- `test11_good.json`: start shell (bash) with `command_line` and `cwd`.

Bad Tests

- test1_bad.json: query uses wrong target and non-namespaced args.
- test2_bad.json: start process with misspelled field and wrong type.
- test3_bad.json: copy artifact with null payload.
- test4_bad.json: stop process with PID as string.
- test5_bad.json: query features as string instead of array.
- test6_bad.json: start process with name as number.
- test7_bad.json: stop process with PID as string.
- test8_bad.json: copy artifact with empty payload and incomplete storage.
- test9_bad.json: delete files with wrong field types.
- test10_bad.json: query features array contains numbers.

Results Summary

Function	Good Tests	Good Passed	Bad Tests	Bad Passed
test_decoding	11	11	0	0
test_encoding	11	11	0	0
test_response_to_invalid_commands	0	0	10	10
test_sending	11	11	0	0
test_sending_invalid	0	0	10	10

Table 8.4: RCLI JSON Validation Summary (53 passed, 0 failed, 124.45s)

8.3.2 NFM Profile

Overview

The NFM tests validate complete JSON round-trips for commands such as starting and stopping network monitors, handling IPv4/IPv6 filters, information elements, and exporter configurations.

Good Tests

- test1_good.json: query features (information_elements, exports, interfaces).
- test2_good.json: start monitor with interface and IPv4/IPv6 filters.

- test3_good.json: start monitor with exporter (storage + collectors).
- test4_good.json: stop monitor using `x-nfm:monitor_id`.
- test5_good.json: start monitor with exporter options and scheduled times.
- test6_good.json: start minimal monitor (interface + IE).
- test7_good.json: start monitor with multiple interfaces and filters.
- test8_good.json: start monitor with IPv4-only DNS filter.
- test9_good.json: start monitor specifying information elements and actuator.
- test10_good.json: stop monitor by ID.
- test11_good.json: start monitor with single collector.
- test12_good.json: stop monitor by ID (`probe_stats`).
- test13_good.json: start IPv6-only monitor.

Bad Tests

- test1_bad.json: wrong profile/target mix (RCLI command in NFM suite).
- test2_bad.json: invalid collector IP.
- test3_bad.json: collector port out of range.
- test4_bad.json: empty interface name.
- test5-6_bad.json: stop uses wrong target (`x-nfm:monitor` instead of `monitor_id`).
- test7-8_bad.json: malformed collector IP and port range violations.
- test9_bad.json: invalid interfaces array type.
- test10_bad.json: invalid L4 protocol in filter.
- test11_bad.json: missing action field.
- test12-13_bad.json: collectors of wrong type or wrong IP family.
- test14_bad.json: unknown actuator key.
- test15_bad.json: `monitor_id` provided as object instead of string.
- test16_bad.json: non-CIDR IPv4 filters.
- test17_bad.json: empty collectors array.

Results Summary

Function	Good Tests	Good Passed	Bad Tests	Bad Passed
test_decoding	13	13	0	0
test_encoding	13	13	0	0
test_response_to_invalid_commands	0	0	17	17
test_sending	13	13	0	0
test_sending_invalid	0	0	17	17

Table 8.5: NFM JSON Validation Summary (73 passed, 0 failed, 10.65s)

8.3.3 FCLM Profile

Good Tests

- test1_good.json: query features (export_fields, exports_config, import_controls, imports_config).
- test2_good.json: start file monitor (path pattern) with exporter and import controls.
- test3_good.json: start file monitor with scheduled times.
- test4_good.json: start minimal file monitor (path only).
- test5_good.json: stop monitor by x-fclm:monitor_id.
- test6_good.json: start URI monitor.
- test7_good.json: start socket monitor (UDP syslog).
- test8_good.json: query standard features.
- test9_good.json: start file monitor with collectors and exporter.
- test10_good.json: stop monitor by monitor_id (agent_syslog).
- test11_good.json: query extended features.

Bad Tests

- test1_bad.json: invalid monitor key “files” and malformed path.
- test2_bad.json: exporter uses wrong key “storages”.
- test3_bad.json: collector with invalid IP address.
- test4_bad.json: wrong target key for stop.

- test5_bad.json: unsupported monitor type “artifact”.
- test6_bad.json: URI monitor missing required fields.
- test7_bad.json: socket monitor with negative port.
- test8_bad.json: invalid collector format value.
- test9_bad.json: stop with monitor_id as object instead of string.
- test10_bad.json: export_fields wrong type (number instead of array).

Results Summary

Function	Good Tests	Good Passed	Bad Tests	Bad Passed
test_decoding	11	11	0	0
test_encoding	11	11	0	0
test_response_to_invalid_commands	0	0	10	10
test_sending	11	11	0	0
test_sending_invalid	0	0	10	10

Table 8.6: FCLM JSON Validation Summary (53 passed, 0 failed, 9.84s)

8.3.4 Implementation and Interoperability Considerations

I implemented the RCLI, NFM, and FCLM profiles end-to-end, including the profile data models, actuator logic, and the full validation and testing workflow. Across all three profiles, the primary challenges were not related to command intent or business logic, but rather to interoperability between runtime representations, JSON fixtures, and contributed schema artifacts.

For **RCLI**, the main limitation concerned schema interoperability. While the profile produced semantically correct capability data for runtime behavior, strict validation against the contributed JSON Schema expected a narrower structural representation for certain capability fields. This resulted in validation failures despite correct semantics. To address this, I introduced a normalization step in the validation path. The runtime model remains unchanged, but schema validation operates on a schema-aligned representation derived from the runtime object. This preserves functional correctness while ensuring strict schema compliance.

For **NFM**, the primary limitation was representation variance across semantically equivalent inputs. Valid external payloads appeared in multiple JSON shapes (e.g., alias field names or alternative representations of information elements), while typed decoding logic initially expected a single canonical structure. To resolve this, I implemented compatibility decoding within the NFM profile. The decoder now

accepts alternative valid external forms and normalizes them into a unified internal representation, ensuring both robustness and backward compatibility.

For **FCLM**, the main issue was conformance fragility caused by profile–schema misalignment. Certain contributed schema definitions and emitted payload forms were not perfectly aligned in terms of field naming, casing, structural shape, or reference consistency. This led to validation failures even when the command semantics were correct. To mitigate this, I introduced a schema-facing normalization and patch layer within the validation workflow. This layer adapts emitted payloads to the exact schema expectations without altering runtime semantics, thereby preserving correctness while ensuring conformance.

Overall, these challenges highlight that the most significant complexity in profile implementation lies not in core command semantics, but in maintaining strict interoperability between runtime models, JSON encodings, and externally maintained schema definitions.

8.4 Performance Evaluation

Performance evaluation assesses the responsiveness and efficiency of the implemented OpenC2 profiles under realistic execution conditions. Each test measures the time required for a complete command-response exchange between the *controller* (producer) and the *server* (consumer) over HTTP/JSON, decomposed into distinct phases: controller-side (encoding, send/receive, decoding) and server-side (decoding, processing, encoding).

All experiments were executed locally on the development environment using Flask/Werkzeug servers with the OpenC2 test harness. Although the results reflect single-run executions, they provide a clear picture of relative latency across different command types and profiles.

8.4.1 RCLI Profile

Command	Encode+Hdrs	Send/Recv	Decode	Total
Start Processes	0.0004	0.0068	0.0016	0.0088
Stop Processes	0.0005	0.0093	0.0017	0.0115
Delete Files	0.0005	0.0110	0.0016	0.0132
Query Features	0.0005	0.0072	0.0025	0.0101
Copy Artifact	0.0010	1.3760	0.0033	1.3803

Table 8.7: RCLI Controller-side latency (seconds)

Command	Decoding	Processing	Encoding	Total
Start Processes	0.0006	0.0014	0.0002	0.0022
Stop Processes	0.0006	0.0041	0.0002	0.0049
Delete Files	0.0006	0.0054	0.0003	0.0062
Query Features	0.0006	0.0014	0.0004	0.0024
Copy Artifact	0.9150	0.4540	0.0008	1.3690

Table 8.8: RCLI Server-side latency (seconds)

Large File Transfer Observation. To evaluate data-transfer behavior, a large artifact was copied via the `copy` command. The total controller-side latency was 1.3760 seconds, while the corresponding server-side processing totaled 1.3690 seconds. Given that the operation involves full file I/O and payload handling, this execution time is considered acceptable under local test conditions. The majority of the delay is attributable to file read/write operations rather than OpenC2 encoding or HTTP overhead.

8.4.2 NFM Profile

Action Target	Encode+Hdrs	Send/Recv	Decode	Total
Start → nProbe	0.0008	3.0227	0.0020	3.0254
Start → Packetbeat	0.0005	3.0378	0.0021	3.0404
Stop session	0.0004	0.0102	0.0045	0.0151
Query features	0.0004	0.0204	0.0051	0.0259

Table 8.9: NFM Controller-side latency (seconds)

Action Target	Decoding	Processing	Encoding	Total
Start → nProbe	0.0015	3.0147	0.0004	3.0166
Start → Packetbeat	0.0020	3.0301	0.0003	3.0324
Stop session	0.0008	0.0031	0.0004	0.0043
Query features	0.0006	0.0130	0.0009	0.0145

Table 8.10: NFM Server-side latency (seconds)

Startup Delay Interpretation. The `start` commands targeting nProbe and Packetbeat show a total latency of approximately 3 seconds. However, this duration intentionally includes a 3-second stabilization delay introduced before returning

the success response. When subtracting this delay, the actual OpenC2 actuator execution overhead is on the order of a few milliseconds, comparable to other lightweight commands.

8.4.3 FCLM Profile

Command	Encode+Hdrs	Send/Recv	Decode	Total
Start → Filebeat	0.0005	3.0320	0.0022	3.0347
Stop session	0.0004	0.0058	0.0015	0.0076
Query Features	0.0004	0.0081	0.0050	0.0134

Table 8.11: FCLM Controller-side latency (seconds)

Command	Decoding	Processing	Encoding	Total
Start → Filebeat	0.0010	3.0249	0.0004	3.0262
Stop session	0.0006	0.0011	0.0002	0.0020
Query Features	0.0011	0.0014	0.0005	0.0030

Table 8.12: FCLM Server-side latency (seconds)

Filebeat Startup Interpretation. Similarly, the `start` command for Filebeat includes the same intentional 3-second initialization delay to ensure service readiness. Excluding this delay, the effective command-processing time remains within millisecond range, demonstrating negligible protocol overhead.

8.4.4 Measurement Context and Interpretation

Performance and latency values depend strongly on what operation is performed: the size and location of copied files, the type and number of processes started or stopped, and the system or network load at runtime. Therefore, the following results should be regarded as indicative benchmarks obtained under controlled test conditions rather than absolute performance figures.

Chapter 9

Conclusion

This thesis investigated the design, implementation, and validation of three OpenC2 actuator profiles for remote command and control of security functions: the Remote Command-Line Interface (RCLI), the Network Flow Monitor (NFM), and the File Collection Log Monitoring (FCLM) profiles. Each profile extends the standard OpenC2 language to support practical monitoring and control operations in modern computing environments, while preserving interoperability across heterogeneous systems.

The developed profiles successfully completed both data type and JSON validation suites, ensuring formal correctness and compliance with the OpenC2 specification. Minor discrepancies were observed in some type definitions or encoding tests, mainly due to stricter schema validation and version mismatches between the core and profile definitions. Performance evaluations demonstrated acceptable latency and responsiveness: local commands executed within a few milliseconds, while more complex operations, such as remote file transfers, network monitoring, or log collection, exhibited predictable and bounded delays. These results confirm that the profiles introduce minimal control-plane overhead and can be reliably employed for practical remote command and control scenarios.

The relevance of this work lies in the generality and broad applicability of the implemented profiles. The RCLI profile provides a flexible interface for managing arbitrary shell-based security functions, NFM enables comprehensive network flow monitoring, and FCLM supports structured log collection and export. Together, these profiles cover a wide range of typical monitoring operations, from file and network monitoring to remote command execution, although system-level metrics were not explicitly addressed. Enforcement mechanisms, such as the SLPF policy, were considered in previous work (Catenaro), providing a foundation for secure operation.

Future work may include: 1) extending the set of actuators to cover additional monitoring sources such as eBPF-based tools or system metrics, and 2) integrating

the existing fine-grained Authentication, Authorization, and Accounting (AAA) mechanisms with the developed profiles, enabling secure and user-aware control of OpenC2 commands. These directions would further enhance the scope, security, and practical applicability of the profiles in complex environments.

Overall, this thesis demonstrates that OpenC2 profiles can be effectively designed and deployed to control diverse security functions, providing a uniform, extensible, and low-overhead framework for automation, monitoring, and orchestration in modern cybersecurity systems.

Appendix A

OpenC2 Profile Command Examples

This appendix collects the detailed OpenC2 command and response examples used to validate and demonstrate the behavior of the implemented profiles. Each section corresponds to one Actuator Profile developed in this work.

A.1 Remote Command-Line Interface Profile

A.1.1 Features (Query)

Type: Features (ArrayOf(Feature))

Example 1 – Querying All Supported Features

```
{
  "action": "query",
  "target": {
    "features": ["versions", "profiles", "pairs", "
    clicommands"]
  },
  "args": {
    "response_requested": "complete"
  }
}
```

Listing A.1: Command – Querying All Supported Features

```
{
  "status": 200,
  "status_text": "OK",
```

```
"results": {
  "versions": ["1.0"],
  "profiles": ["x-rcli"],
  "pairs": {
    "query": ["features", "x-rcli:processes", "x-rcli:
files"],
    "copy": ["artifact"],
    "start": ["x-rcli:processes"],
    "stop": ["x-rcli:processes"],
    "delete": ["x-rcli:files"]
  },
  "clicommands": [
    {"name": "ls", "command_line": "-l -a -h"},
    {"name": "cat", "command_line": "<file>"},
    {"name": "rm", "command_line": "<path> -r -f"}
  ]
}
```

Listing A.2: Response – Querying All Supported Features

A.1.2 Artifact (Copy)

Type: Artifact (Record)

Example 1 – Copy Artifact

```
{
  "action": "copy",
  "target": {
    "artifact": {
      "mime_type": "application/x-sh",
      "payload": {
        "uri": "https://example.com/job.sh"
      }
    }
  },
  "args": {
    "storage": {
      "path": "/opt/data",
      "name": "job.sh"
    }
  }
}
```

Listing A.3: Command – Copy Artifact

```
{
  "status": 200,
  "results": {
    "file_status": [
      {"path": "/opt/data", "name": "job.sh"}
    ]
  }
}
```

Listing A.4: Response – Copy Artifact

A.1.3 x-rcli:processes (Start, Stop, Query)

Type: Process (ArrayOf(Process))

Example 1 – Start Process

```
{
  "action": "start",
  "target": {
    "x-rcli:processes": [
      {"name": "ls", "command_line": "-la /tmp"}
    ]
  }
}
```

Listing A.5: Command – Start Process

```
{
  "status": 200,
  "results": {
    "process_status": [
      {"pid": 1234, "state": "running"}
    ]
  }
}
```

Listing A.6: Response – Start Process

Example 2 – Query Processes

```
{
  "action": "query",
  "target": {"x-rcli:processes": []}
}
```

Listing A.7: Command – Query Processes

```
{
  "status": 200,
  "results": {
    "process_status": [
      {"pid": 1234, "state": "running"},
      {"pid": 5678, "state": "sleeping"}
    ]
  }
}
```

Listing A.8: Response – Query Processes

Example 3 – Stop Process

```
{
  "action": "stop",
  "target": {
    "x-rcli:processes": [{"pid": 1234}]
  }
}
```

Listing A.9: Command – Stop Process

```
{
  "status": 200,
  "status_text": "OK"
}
```

Listing A.10: Response – Stop Process

A.1.4 x-rcli:files (Query, Delete)

Type: File (ArrayOf(File))

Example 1 – Query Files

```
{
  "action": "query",
  "target": {"x-rcli:files": []}
}
```

Listing A.11: Command – Query Files

```
{
  "status": 200,
  "results": {
    "file_status": [
      {"path": "/opt/data", "name": "script.sh"},
      {"path": "/opt/data", "name": "config.json"}
    ]
  }
}
```

Listing A.12: Response – Query Files

Example 2 – Delete File

```
{
  "action": "delete",
  "target": {
    "x-rcli:files": [
      {"path": "/opt/data", "name": "script.sh"}
    ]
  }
}
```

Listing A.13: Command – Delete File

```
{
  "status": 200,
  "status_text": "OK"
}
```

Listing A.14: Response – Delete File

A.2 NFM Command and Response Examples

A.2.1 Features (Query)

Type: Features (ArrayOf(Feature))

Example 1 – Query Features

```
{
  "action": "query",
  "target": {
    "features": [
      "versions", "profiles", "pairs",
      "interfaces", "information_elements", "exports"
    ]
  },
  "args": {
    "response_requested": "complete"
  }
}
```

Listing A.15: Command – Query Features

```
{
  "status": 200,
  "status_text": "OK",
  "results": {
    "versions": ["1.0"],
    "profiles": ["x-nfm"],
    "pairs": {
      "query": ["features"],
      "start": ["x-nfm:monitor"],
      "stop": ["x-nfm:monitor_id"]
    },
    "interfaces": [
      {"name": "eth0", "active": "UP"},
      {"name": "eth1", "active": "DOWN"}
    ],
    "information_elements": ["source ip", "destination ip",
      "bytes", "packets"],
    "exports": ["file", "collector"],
    "export_options": ["sampling", "aggregate", "format"]
  }
}
```

Listing A.16: Response – Query Features

A.2.2 x-nfm:monitor (Start)

Type: Monitor (Record)

Example 1 – Start Monitor

```
{
  "action": "start",
  "target": {
    "x-nfm:monitor": {
      "interfaces": [{"name": "eth0"}],
      "information_elements": ["source ip", "destination ip"],
      "filter_v4": [
        {"src_addr": "192.168.1.0/24", "protocol": "tcp", "dst_port": 443}
      ]
    }
  },
  "args": {
    "exporter": {
      "collectors": [{"address": "10.0.0.10/32", "port": 2055}]
    },
    "exporter_options": {"format": "json", "sampling": 20},
    "response_requested": "complete"
  }
}
```

Listing A.17: Command – Start Monitor

```
{
  "status": 200,
  "results": {
    "monitor_id": "Alpha_7FS9PZ",
    "status_text": "Monitoring started"
  }
}
```

Listing A.18: Response – Start Monitor

Example 2 – Schedule Monitor Start

```
{
  "action": "start",
  "target": {
    "x-nfm:monitor": {
```

```
    "interfaces": [{"name": "eth1"}],
    "information_elements": ["source ip", "destination ip"]
  }
},
"args": {
  "start_time": "2025-05-10T10:00:00Z",
  "stop_time": "2025-05-10T10:15:00Z",
  "response_requested": "complete"
}
}
```

Listing A.19: Command – Schedule Monitor Start

```
{
  "status": 102,
  "status_text": "Monitoring scheduled",
  "results": {
    "monitor_id": "Bravo_2K4MRX"
  }
}
```

Listing A.20: Response – Schedule Monitor Start

A.2.3 x-nfm:monitor_id (Stop, Query)

Type: MonitorID (Identifier)

Example 1 – Stop Monitor

```
{
  "action": "stop",
  "target": {
    "x-nfm:monitor_id": "Alpha_7FS9PZ"
  },
  "args": {
    "response_requested": "complete"
  }
}
```

Listing A.21: Command – Stop Monitor

```
{
  "status": 200,
  "status_text": "Process terminated"
}
```

Listing A.22: Response – Stop Monitor

Example 2 – Query Active Monitors

```
{  
  "action": "query",  
  "target": {"x-nfm:monitor_id": []}  
}
```

Listing A.23: Command – Query Active Monitors

```
{  
  "status": 200,  
  "results": {  
    "active_monitors": [  
      {"monitor_id": "Alpha_7FS9PZ", "state": "running"},  
      {"monitor_id": "Bravo_2K4MRX", "state": "scheduled"}  
    ]  
  }  
}
```

Listing A.24: Response – Query Active Monitors

A.3 File Collection and Log Monitoring Profile

A.3.1 Features (Query)

Type: Features (ArrayOf(Feature))

Example 1 – Query FCLM Features

```
{
  "action": "query",
  "target": {
    "features": [
      "versions", "profiles", "pairs",
      "export_fields", "exports_config",
      "imports_config", "import_controls"
    ]
  },
  "args": {
    "response_requested": "complete"
  }
}
```

Listing A.25: Command – Query FCLM Features

```
{
  "status": 200,
  "status_text": "OK",
  "results": {
    "versions": ["1.0"],
    "profiles": ["x-fclm"],
    "pairs": {
      "query": ["features"],
      "start": ["x-fclm:monitor"],
      "stop": ["x-fclm:monitor_id"]
    },
    "x-fclm": {
      "export_fields": ["timestamp", "metadata"],
      "exports_config": [
        "storage:path",
        "collector:address/port/format"
      ],
      "imports_config": [
        "scan_frequency",
        "max_backoff",
        "close_inactive"
      ]
    }
  }
}
```

```

    "import_controls": [
      "scan_frequency",
      "max_backoff",
      "ignore_older"
    ]
  }
}

```

Listing A.26: Response – Query FCLM Features

A.3.2 x-fclm:monitor (Start)

Type: LogMonitor (Record)

Example 1 – Start File Monitoring

```

{
  "action": "start",
  "target": {
    "x-fclm:monitor": {
      "file": {"path": "/var/log/*.log"}
    }
  },
  "args": {
    "response_requested": "complete",
    "x-fclm": {
      "log_exporter": {
        "storage": {"path": "/var/log/filebeat/flows"},
        "collectors": [
          {"address": "192.1.1.6/32", "port": 1234, "format": "json"}
        ]
      },
      "import_controls": {"scan_frequency": 1000, "max_backoff": 10},
      "export_fields": ["timestamp", "metadata"]
    }
  },
  "actuator": {"x-fclm": {"asset_id": "agent_x"}}
}

```

Listing A.27: Command – Start File Monitoring

```

{

```

```
"status": 200,
"results": {
  "x-fclm": {
    "monitor_id": "Theta_7197KK",
    "export_fields": ["timestamp", "metadata"]
  }
}
```

Listing A.28: Response – Start File Monitoring

Example 2 – Start Socket Monitoring

```
{
  "action": "start",
  "target": {
    "x-fclm:monitor": {
      "socket": {
        "host": "192.168.1.20/32",
        "port": 514,
        "protocol": "UDP"
      }
    }
  },
  "args": {
    "response_requested": "complete",
    "x-fclm": {
      "export_fields": ["timestamp", "payload"],
      "log_exporter": {
        "collectors": [
          {"address": "10.0.0.5/32", "port": 5044, "format":
"json"}
        ]
      }
    }
  }
}
```

Listing A.29: Command – Start Socket Monitoring

```
{
  "status": 200,
  "results": {
    "x-fclm": {"monitor_id": "Delta_2251PX"}
  }
}
```

Listing A.30: Response – Start Socket Monitoring

A.3.3 x-fclm:monitor__id (Stop)

Type: MonitorID (Identifier)

Example 1 – Stop Monitoring

```
{
  "action": "stop",
  "target": {"x-fclm:monitor_id": "Theta_7197KK"},
  "args": {"response_requested": "complete"}
}
```

Listing A.31: Command – Stop Monitoring

```
{
  "status": 200,
  "status_text": "OK"
}
```

Listing A.32: Response – Stop Monitoring

Example 2 – Scheduled Stop

```
{
  "action": "stop",
  "target": {"x-fclm:monitor_id": "Delta_2251PX"},
  "args": {
    "stop_time": "2025-05-10T10:30:00Z",
    "response_requested": "complete"
  }
}
```

Listing A.33: Command – Scheduled Stop

```
{
  "status": 200,
  "status_text": "Scheduled termination"
}
```

Listing A.34: Response – Scheduled Stop

Bibliography

- [1] Vasileios Mavroeidis and Joe Brule. «A nonproprietary language for the command and control of cyber defenses – OpenC2». In: *Computers & Security* 97 (2020), p. 101999. ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2020.101999>. URL: <https://www.sciencedirect.com/science/article/pii/S0167404820302728> (cit. on pp. 1, 2).
- [2] National Institute of Standards and Technology. *The Cybersecurity Framework 2.0*. Tech. rep. NIST Special Publication 800-55 Draft references OpenC2 under automated response. U.S. Department of Commerce, 2024. URL: <https://www.nist.gov/cyberframework> (cit. on p. 1).
- [3] Robert Barnes and Patrick Nguyen. «Security Orchestration and Automated Response: Interoperability Challenges and OpenC2 Alignment». In: *IEEE International Conference on Cyber Security and Resilience*. 2021, pp. 147–154. DOI: 10.1109/CSR51186.2021.9527973 (cit. on p. 2).
- [4] N. Kheir and O. Dib. «Semantic Consistency in Cyber Defense Command Languages: A Comparative Study». In: *Computers & Security* 109 (2021), p. 102393. DOI: 10.1016/j.cose.2021.102393 (cit. on p. 2).
- [5] National Security Agency and Johns Hopkins University Applied Physics Laboratory. *Integrated Adaptive Cyber Defense (IACD) Framework*. Tech. rep. JHUAPL, 2018. URL: <https://www.iacdautomate.org/framework> (cit. on p. 2).
- [6] Markus Miettinen and Lei Zhao. «Automated Cyber Defense in Distributed Cloud Infrastructures Using OpenC2». In: *Journal of Network and Computer Applications* 204 (2022), p. 103414. DOI: 10.1016/j.jnca.2022.103414 (cit. on p. 2).
- [7] Yue Zhang, Joe Brule, and Tao Li. «Automated Threat Response Using STIX/TAXII and OpenC2». In: *Proceedings of the ACM Workshop on Automated Decision Making for Active Cyber Defense*. 2020, pp. 11–18. DOI: 10.1145/3428471.3428483 (cit. on p. 2).

- [8] David Lemire (ed.) and David Kemp (ed.) *OpenC2 Actuator Profile Development Process Version 1.0*. OASIS Committee Note 01. Jan. 2024. URL: <https://docs.oasis-open.org/openc2/cn-appdev/v1.0/cn01/cn-appdev-v1.0-cn01.html> (cit. on pp. 3, 6, 9).
- [9] David Lemire (ed.) *Specification for Transfer of OpenC2 Messages via HTTPS Version 1.1*. OASIS Committee Specification 01. Nov. 2021. URL: <https://docs.oasis-open.org/openc2/open-impl-https/v1.1/cs01/open-impl-https-v1.1-cs01.html> (cit. on pp. 3, 7, 14).
- [10] David Lemire (ed.) *Specification for Transfer of OpenC2 Messages via MQTT Version 1.0*. OASIS Committee Specification 01. Nov. 2021. URL: <https://docs.oasis-open.org/openc2/transf-mqtt/v1.0/cs01/transf-mqtt-v1.0-cs01.html> (cit. on pp. 3, 7).
- [11] Jason Romano (ed.) and Duncan Sparrell (ed.) *Open Command and Control (OpenC2) Language Specification Version 1.0*. OASIS Committee Specification 02. Nov. 2019. URL: <https://docs.oasis-open.org/openc2/oc2ls/v1.0/cs02/oc2ls-v1.0-cs02.html> (cit. on pp. 3, 6, 11–13).
- [12] Duncan Sparrell (ed.) *Open Command and Control (OpenC2) Architecture Specification Version 1.0*. OASIS Committee Specification 01. Sept. 2022. URL: <https://docs.oasis-open.org/openc2/oc2arch/v1.0/cs01/oc2arch-v1.0-cs01.html> (cit. on p. 5).