

Flexible and extensible implementation of OpenC2

Matteo Repetto
IMATI, CNR
Genoa, Italy
matteo.repetto@ge.imati.cnr.it

Enrico Cambiaso
IEIIT, CNR
Genoa, Italy
enrico.cambiaso@cnr.it

Abstract—Recent management paradigms for software-defined infrastructures bring more agility in the creation and operation of digital services, but also introduce new cyber-security issues due to fast-changing environments and dynamic topologies. Rigid and statically-configured architectures are no more suitable for managing cyber-security functions in mixed cloud/6G/IoT environments, since the number, capabilities, and providers of such functions are expected to change at run-time. Recently, OpenC2 has been proposed as common interface to orchestrate heterogeneous and multi-vendor security functions in an homogeneous way.

In this paper we describe our open-source implementation of OpenC2. Our software is explicitly designed to be portable, extensible, and easy to use for both users and developers. The main objective is to simplify the addition of new profiles as soon as they are available, as well as the usage of different transport protocols and encoding formats. At the best of our knowledge, our code represents the first and more ambitious effort to put OpenC2 specification into work.

I. INTRODUCTION

The evolution of 5G networks into a large, pervasive, and powerful computing continuum is radically changing the way digital services are created and operated. Service-oriented architectures and software-defined infrastructures boost new management models, where digital services are composed by chaining software-defined resources and functions from heterogeneous domains: Network Function Virtualization (NFV), cloud/edge/fog applications, Internet of Things (IoT), and data [1], [2]. They also continuously evolve at run-time, according to human-defined orchestration rules or, in perspective, even cognitive Artificial Intelligence (AI) processes. Effective protection of such systems against cyber-threats becomes more challenging, due to the lack of a sharp and effective security perimeter, the usage of third parties' infrastructures and services, and the introduction of serverless computing. A transition towards adaptive and agile cyber-defense architectures is therefore necessary beyond existing models, where most cyber-security appliances are manually configured for static environments and work in isolation [3], which eventually delays mitigation and response actions. The ultimate goal is homogeneous, seamless, and transparent orchestration of capillary and programmable monitoring, detection, and enforcement capabilities at the edge and in user devices [1].

Standard interfaces to security functions would simplify the composition of monitoring, detection, and enforcement processes over heterogeneous service chains deployed in mixed 5G/6G/cloud/IoT infrastructures, removing the need

for inflexible and hardly manageable adapters, as commonly happens with Cloud Access Security Brokers (CASBs). In this respect, Open Command and Control (OpenC2) [3] was proposed by OASIS as both an abstract model of security capabilities and a common communication protocol to security functions. However, despite of the large community that backs its definition, open-source implementations of OpenC2 are still missing. We argue that this hinders its adoption, as well as its improvement with additional features in terms of both function profiles and additional capabilities.

In this paper, we describe our implementation of an OpenC2 library, which source code is publicly available for free. We designed our software to be it easily extensible, so that additional security functions can be supported as soon as new profiles are defined, without the need to dig deeper into the code. Additional encoding formats and transport protocols can be used, to satisfy multiple use cases. Finally, integration with external security orchestrators is straightforward, by using the library itself of by building custom interfaces and bindings.

The rest of the paper is organized as follows. First, we give a brief overview of Related Work in Sec. II. Then, we describe the OpenC2 architecture and language specification in Sec. III. Afterwards, we describe the software architecture of our OpenC2 library in Sec. IV and give concrete examples of its usage and extension in Sec. V. Finally, we give our conclusion in Sec. VI.

II. RELATED WORK

The need to discover and configure remote security agents in large computing environments has been a recurring issue. In the SECURED project [4], high-level policies are translated into specific configurations, similarly to CASBs; unfortunately, this approach does not scale well with heterogeneous interfaces, because a different “adapter” or “driver” should be provided for each of them.

A similar approach was followed by Bondan et al. [5], who use an orchestrator abstraction driver to retrieve information about running Virtual Network Functions (VNFs) and to notify anomalies. They leverage the Management and Orchestration (MANO) NFV orchestrator to retrieve VNF identifiers, network, Service Function Chaining (SFC) endpoints, connection links, and network configuration (e.g., bandwidth). The scope is limited to SFC status and network configuration, hence leaving out any form of integrity verification and network attack.

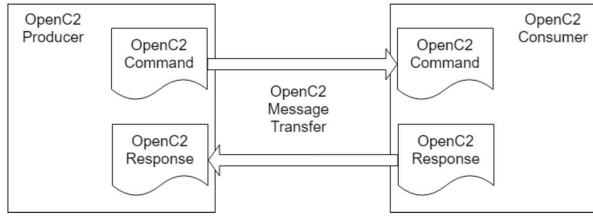


Fig. 1. OpenC2 architecture.

A similar approach is followed by Repetto *et al.* [6], who also introduce an abstraction layer to discover and configure capabilities of security agents. Carrega *et al.* [7] implemented the GUARD framework for remote management of cyber-security functions and dynamic composition of monitoring, detection, and response processes for digital service chains; in this case, a Smart Data Model (SDM) is used to discover their location, to describe their capabilities, and to configure them.

An overview of OpenC2 was originally given by Mavroeidis and Brule [3]. Despite of the large number of software resources available¹, there is currently no library which can be easy to use, to integrate in existing software, and to extend with additional profiles.

III. OPEN COMMAND AND CONTROL

OpenC2 was born to make coordination in cyber-relevant time between decoupled blocks that perform cyber-defense functions. The purpose is to support technology diversity, which undeniably introduces an extra layer of security, but with standardized function-centric interfaces that make security automation and orchestration feasible and less complex to achieve. OpenC2 is focused on Machine-to-Machine (M2M) communication for cyber-defense components and only implements the Acting part of the Integrated Adaptive Cyber Defense (IACD) framework, hence leaving sensing, analytics, and decision-making out of scope.

A. OpenC2 architecture

As the name suggests, OpenC2 is designed to send commands to remote cyber-defense appliances. The communication happens according to a client/server model, where a *Producer* sends *commands* to a *Consumer*, which executes them; the same entity can play both roles [8]. In response to commands, Consumers may generate a *response*, which carries the result of executing the command, as shown in Fig. 1.

An OpenC2 Consumer may implement a single or multiple security functions; in the last case the function should be explicitly identified in the command, otherwise it is executed for

all functions that support it. An OpenC2 Consumer may also act as a proxy towards networked security functions, either controlled by OpenC2 itself or another protocol. The proxy architecture may be useful to manage protocol conversions.

B. OpenC2 language

The language specification [9] defines the common syntax for OpenC2 payloads, in a transfer-agnostic way. In this respect, there are two distinct payload structures: *Command* and *Response*.

A Command (top of Listing 1) contains the following elements:

- **Action:** the instruction, task or activity to be performed (e.g., start, stop, locate, set, update, create).
- **Target:** the object of the action, which is the logical entity affected by the execution of the instruction (e.g., a file, a domain name, an email, a feature, a network connection, a device).
- **Arguments:** additional information on how the command is performed (e.g., time interval, duration, periodicity).
- **Actuator:** the entity that executes the action (e.g., firewall).

Both Targets and Actuators can be specified with different levels of granularity, namely they may identify either a specific object, or a list or a group of objects.

The syntax elements of OpenC2 follows typical language patterns, with a subject (Actuator), a verb (Action), an object (Target), and complements (Arguments). An action-target pair defines a command.

The OpenC2 Language Specification defines a *standard* set of actions and a *baseline* collection of targets for those actions. This means that the set of actions must be the same for every implementation and limited to what defined in the Language Specification; no additions are allowed for each specific Actuator. In contrast, the set of targets in the Language Specification is extensible by definition, and each cybersecurity function is allowed to identify its own target, if necessary. This approach is meant to balance the need for commonality and interoperability of implementations, while recognizing the diversity of cybersecurity functions.

Each element in OpenC2 Commands and Responses is defined in terms of primitive types (e.g., binary, integer, string) and derived structures (e.g., array, map, enumerated). Structures are recursively used and coupled with semantic constraints to derive typical data objects, as IP/MAC addresses, emails, domain names, dates and times, digests, etc.

A Response (bottom of Listing 1) contains the following elements:

- **Status:** An integer that represents the exit status of the command execution.
- **Status text:** A human-readable description of the status.
- **Result:** a list of key/value pairs produced by the execution of the command.

Even though the OpenC2 language is transfer-agnostic, status code values substantially follow those already defined for Hyper-Text Transfer Protocol (HTTP).

¹OpenC2 Open Source Software, URL: <https://openc2.org/opensource.html>.

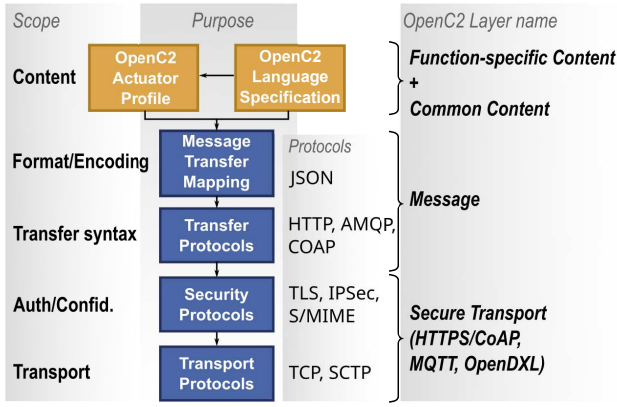


Fig. 2. OpenC2 layering model. Orange boxes falls under the scope of OpenC2, blue boxes are existing protocols.

Serialization is necessary for interoperability with transfer protocols. The language specification demands the support for JavaScript Object Notation (JSON) serialization and defines the corresponding data type mapping.

C. OpenC2 stack

The OpenC2 communication stack is layered as shown in Fig. 2, which shows the scope, purpose, main protocols, and name for each layer. At the top of the stack, the Content layer includes both common and function-specific language elements, as defined by the *language specification* [9] and *actuator profiles* [10], respectively. The Message layer defines how OpenC2 content is encoded (currently, as JSON² objects) and encapsulated into different transfer protocols (e.g., HTTP, CoAP, AMQP). The *transfer specification* (e.g., OpenC2 over HTTPS [11]) maps metadata and content elements into specific protocol elements (e.g., HTTP header and body, respectively). Finally, Secure Transport conveys data in a secure way over common Internet protocols (e.g., TLS/TCP).

An OpenC2 message is made of a header and a payload (also indicated as “message body,” see Listing 1). The header is composed of metadata, including content type, message type, status, request identifier, creation date, origin and destination; it is assembled at the Message layer as defined by the transfer specification. The payload bears the OpenC2 content within an “openc2” object, and it follows the syntax defined in the OpenC2 language. A relevant example of transfer protocol is the HTTP: OpenC2 commands are sent with the POST method to a well known path (`/.well-known/openc2`), and the header and payload are carried in the body of the HTTP message.

D. OpenC2 profiles

The list of Actions provided by the language specification accounts for a broad range of discovery, control, and manipulation operations. However, each Action can only be meaningfully applied to a subset of Targets and implemented

²JSON is another human- and machine readable text format for storing and transporting data.

```
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": "deny",
          "target": {
            "ipv4_connection": {
              "protocol": "tcp",
              "src_port": 21
            }
          }
        },
        "args": {
          "slpf": {
            "drop_process": "none",
            "direction": "ingress"
          }
        },
        "actuator": {
          "slpf": {}
        }
      }
    }
  },
  "body": {
    "openc2": {
      "response": {
        "status": 200,
        "status_text": "Rule correctly inserted"
      }
    }
  }
}
```

Listing 1: Example of OpenC2 command and response for denying inbound FTP connections.

by specific Actuators. Moreover, maintaining a common syntax for the great heterogeneity of security functions and their implementations is largely impracticable. Actuator profiles define semantic constraints and language extensions for specific cyber-defense functions; currently, there are several working documents for different appliances, but only the profile for stateless packet filtering has been formally published [10].

An OpenC2 profile defines which combinations of Actions and Targets are relevant for a specific Actuator, including the corresponding Arguments, in the form of a *command matrix*. A profile may also include extensions to the basic syntax, which account for specific features and characteristics of the

Actuator. For instance, the Stateless Packet Filtering (SLPF) profile includes a “rule number” target which is typically used to update/delete filtering rules, and specific arguments to apply dropping strategies and distinguish between incoming and outgoing packets.

IV. OPENC2 LIBRARY DESIGN

Our OpenC2 library is implemented in Python, which allows good portability across almost any software environment where security functions can be implemented. The choice of Python also brings the full power of reflective programming, which enables to write the code without the need to know in advance any possible data structure and type. This is especially useful to add new profiles, transport protocols, and encoding formats without changing the core of the library.

The library is conceived to dynamically create a working **Openc2!** (**Openc2!**) *stack* for transferring Commands and Responses. An OpenC2 *stack* is designed according to Fig. 2, and includes Content, Message Transfer Mapping, and Transfer Protocols. We do not explicitly account for Security Protocols and Transport Protocols, since they are commonly included in the implementation of Transfer Protocols themselves.

There are two ways to use the library from the Producer side. Every OpenC2 message can be directly sent to an OpenC2 Consumer, by specifying the full OpenC2 stack each time; this is the most straightforward way to use the library without an existing security controller. Alternatively, an OpenC2 Producer object can be created with its own stack, which remains the same for every Consumers. This is likely preferred when a security controller must be designed from scratch, and could benefit from native OpenC2 integration.

From the Consumer side, an Actuator must be provided to translate a given Profile to the specific implementation of the corresponding security function. For instance, an Actuator may translate SLPF commands to `iptables` instructions.

Fig. 3 shows the overall architecture and intended usage of the OpenC2 library. We define specific data structures for all language elements, starting from the Command/Response and recursively down to all their building objects. We also define the prototypes of the Encoding and Transfer layers, together with their interfaces. The concrete implementations of these interfaces will account for the different encoding formats and transfer protocols; currently, we provide the implementation for json encoding (`JSONEncoder`) and HTTP transfer (`HTTPTransfer`).

Most of the Python files in the library folders define the language elements according to the syntax definition in the Language Specification [9]. In particular, `datatypes.py` contains all intermediary data types used to define Actions, Targets, and Arguments. The full implementation of all OpenC2 types might appear useless, since there are already good and widely-used Python libraries for most of them, and they are indeed used to implement them. However, this approach decouples the OpenC2 Application Programming Interface (API) to the underlying implementation, and allows

future updates of the library without affecting the external software that uses it.

A minor deviation from the Language Specification is represented by Messages: their syntax is not dictated explicitly, and the Language Specification just gives a list of data that should be present. Indeed, the syntax of OpenC2 Messages is transfer-specific, and it is therefore defined for each different Transfer protocol. Our implementation defines an internal structure for Messages, since this is necessary to associate some metadata and carry them around until the protocol-specific messages are created.

The main scientific contribution of our OpenC2 library is perhaps given by the flexible and extensible mechanism for serialization, which comes from two main objectives: i

- 1) to provide a general-purpose framework to serialize all Python objects without the need to create custom and repetitive methods for each of them;
- 2) to support additional serialization formats (e.g., eXtensible Markup Language (XML), YAML Ain't Markup Language (YAML)) with minimal impact on the internal serialization engine.

Our serialization approach is based on an intermediate representation which uses Python *dictionaries*. Indeed, Python dictionaries are simple to be translated in json or other kind of representation; at the same time, every Python object comes with a dictionary representation. The whole serialization process therefore works on two layers. At the top layer, each OpenC2 type defines the rule to convert an instance to a dictionary and vice-versa. At the bottom layer, each Encoder provides specific encoding for dictionaries. The same process is used in the opposite direction for de-serialization: the bottom layer creates dictionaries, which are then used to create object instances at the top layer.

Unfortunately, the dictionary representation of Python objects does not fit the serialization requirements of all OpenC2 data, so specific methods must be added for this purpose. To avoid the cumbersome need to provide redundant methods for each OpenC2 object, we leverage the recursive syntax definition in the Language Specification. In fact, a few primitive data types and structures are defined, together with their serialization requirements (see Sec. 3.1 [9]). This way, OpenC2 elements can be incrementally added both in the core library and in profiles without worrying about their serialization, as far as they are derived from the primitive types.

The only implementation constraint that comes from this approach is that every Python object must strictly matches the terminology and field order required by the corresponding element definition in the Language Specification. This is a minor drawback, since following the terminology of the standard is anyway a good practice to write good code that is easily understandable. However, this is not always possible, because some field names are very common words and may clash with Python keywords (e.g., this happens with the “from” field of OpenC2 Messages). We account for this by trailing an underscore to such field names, which is easy to remove during serialization. Finally, every internal variable which is

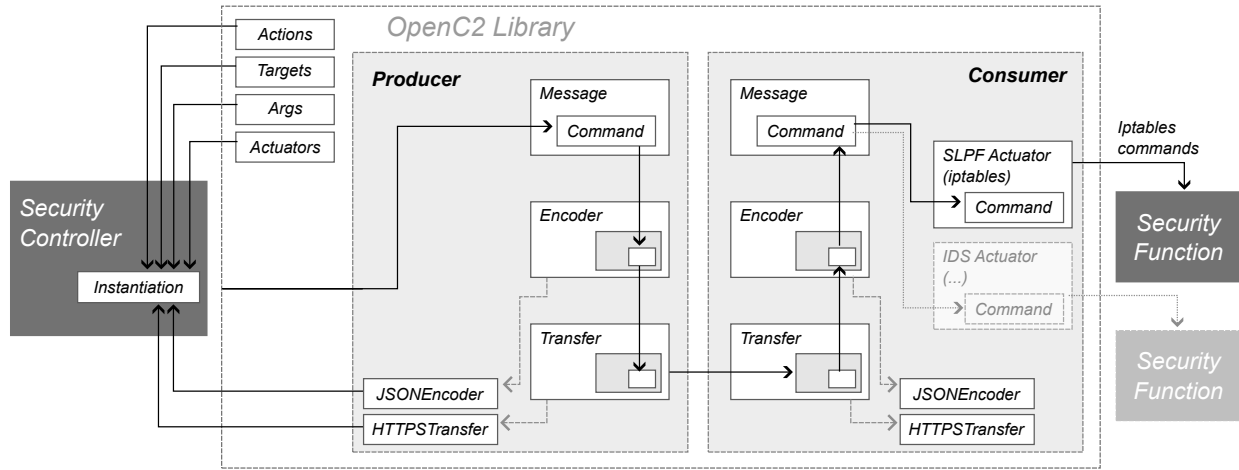


Fig. 3. Overall architecture and workflow of the OpenC2 library

```

from openc2.producer import Producer
from openc2.message import Command
from openc2.encoders.json_encoder import JSONEncoder
from openc2.transfers.http_transfer import HTTPSTransfer

from openc2.actions import Actions
from openc2.targets import *
from openc2.datatypes import IPv4Net, IPv4Connection

# Creating the stack
p = Producer("ge.imati.cnr.ir", JSONEncoder(),
    ↳ HTTPSTransfer("acme.com", 8080))
# Sending Command
resp = p.sendcmd(Command(Actions.scan,
    ↳ IPv4Net("192.168.17.0/24")))
print(resp)
resp = p.sendcmd(Command(Actions.delete,
    ↳ IPv4Connection(dst_addr = "192.168.17.0/24",
    ↳ dst_port=80, protocol=L4Protocol.sctp)))
print(resp)

```

Listing 2: Example of usage of the OpenC2 Producer to send Command and get Response.

not defined in the Language Specification must be kept private, so to exclude it from serialization too.

V. LIBRARY USAGE AND EXTENSIONS

Using the OpenC2 library as a Producer is rather simple. The basic steps include:

- 1) Select an Encoder object for serialization (e.g., JSONEncoder).
- 2) Select a Transfer protocol (e.g., HTTPEncoder).
- 3) Create an OpenC2 Command, including at least Action and Target.
- 4) Send the Command and get back the Response (the send function is blocking).

Listing 2 contains a minimal working example of usage via the Producer object.

On the other side, the Consumer needs the concrete implementation of Actuators for the profile(s) it is serving. The current implementation is still a mock-up, so no response is actually returned so far. Indeed, the whole library is designed to be populated with incremental updates from the community,

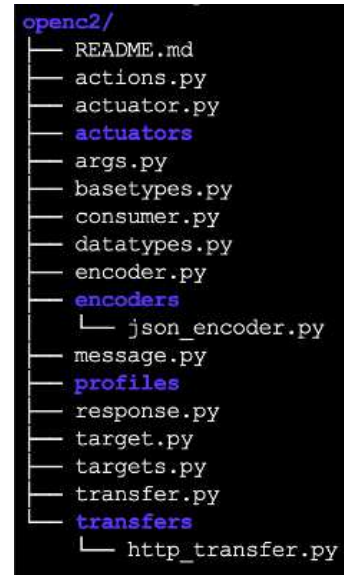


Fig. 4. Structure of the OpenC2 library folder.

as soon as new profiles are defined for security functions and they are applied to different implementations. The same applies to the addition of further Encoders and Transfers.

The structure of the library folder is just organized to allow incremental extensions with additional Profiles, Encoders, and Transfers (see 4). They must be added in the profiles, encoders, and transfers folders, respectively. Similarly, the implementation of Actuators for different security functions will be put in the actuators folder. Since Actuators are always part of a Profile, the presence of a separate folder for them might appear unnecessary. However, Actuators internally implement the interface to a security function, and it is expected multiple Actuators for the same profile to be present for different implementation and technologies (e.g., a stateless packet filter might be implemented with iptables, pf, pfsense, OpenFlow, commercial routers, etc.).

VI. CONCLUSION

In this paper we have presented a Python library that implements the core OpenC2 Language Specification. The main novelty of our work is given by its flexibility and extensibility, which allow incremental addition of Profiles, Encoders, and Transfers, and makes our library suitable to create custom OpenC2 stacks with minimal effort. Although the current code is just a mock-up, it already contains the main skeleton to quickly get to an alpha release in the next months. Future work will complete the implementation with a working HTTPTransfer implementation and an SLPF Actuator for iptables.

REFERENCES

- [1] M. Repetto, A. Carrega, and R. Rapuzzi, “An architecture to manage security operations for digital service chains,” *Future Generation Computer Systems*, vol. 115, pp. 251–266, February 2021.
- [2] L. Cui, F. P. Tso, and W. Jia, “Federated service chaining: Architecture and challenges,” *IEEE Communications Magazine*, no. 3, pp. 47–53, March 2020.
- [3] V. Mavroeidis and J. Brule, “A nonproprietary language for the command and control of cyber defenses – OpenC2,” *Computers & Security*, vol. 97, no. 101999, October 2020.
- [4] D. Montero, M. Yannuzzi, A. L. Shaw, L. Jacquin, A. Pastor, R. Serral-Gracià, A. Lioy, F. Risso, C. Basile, R. Sassu, M. Nemirovsky, F. Ciaccia, M. Georgiades, S. Charalambides, J. Kuusijärvi, and F. Bosco, “Virtualized security at the network edge: a user-centric approach,” *IEEE Communications Magazine*, vol. 53, no. 4, pp. 176–186, April 2015.
- [5] L. Bondan, T. Wauter, B. Volckaert, F. De Turck, and L. Zambenedetti Granville, “NFV anomaly detection: Case study through a security module,” *IEEE Commun. Standards Mag.*, vol. 60, no. 2, pp. 18–24, February 2022.
- [6] M. Repetto, G. Bruno, J. Yusupov, G. Lamanna, B. Ertl, and A. Carrega, “Automating mitigation of amplification attacks in NFV services,” *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2382–2396, September 2022.
- [7] A. Carrega, G. Grieco, D. Striccoli, M. Paooutsakis, T. Lima, J. I. Carretero, and M. Repetto, *Cybersecurity of Digital Service Chains: Challenges, Methodologies and Tools*, ser. Lecture Notes in Computer Science. Springer Nature, April 2022, vol. 13300, ch. A Reference Architecture for Management of Security Operations in Digital Service Chains, pp. 1–31.
- [8] “Open command and control (openc2) architecture,” OASIS standard, September 2022, version 1.0, Committee Specification 01.
- [9] “Open command and control (OpenC2),” OASIS standard, November 2019, language Specification Version 1.0, Committee Specification 02.
- [10] “Open command and control (OpenC2) profile for stateless packet filtering,” July 2019, version 1.0, Committee Specification 01.
- [11] “Specification for transfer of OpenC2 messages via https,” July 2019, version 1.0, Committee Specification 01.