# OpenC2 Library

## Package structure and naming

First of all, the code has been organized in a different way, to make simpler the identification of the different elements, their extension, and their overall reference.

This is the directory tree to be used for this project:

A screen shot of a computer program

Description automatically generated

The main folder will be called “openc2” so all the python code will be available in the openc2 namespace.

The files and folders will be used for the following purposes:

* **actions.py**: this is the list of actions (Sec. 3.3.1.1).
* **actuator.py**: this should contain the base class for the Actuator (Sec. 3.3.1.3).
* **actuators**: definitions of concrete actuators as defined by specific profiles. TODO
* **args.py**: this is the definition of the Args (Sec. 3.3.1.4) (TODO)
* **basetypes**.py: this is the definition of the base components and structures defined by the Language Specification (Sec. 3.1.1); they are mostly intended to take care of the intermediary dictionary translation according to the rules defined in Sec. 3.1. TODO: add missing types.
* **datatypes.py**: this files contains the OpenC2 data types defined throughout the language specification (Sec. 3.4.2)
* **consumer.py**: this is the entry point for receiving commands and sending requests. TODO: defines whether it works in push or pull mode (i.e., with blocking calls or notifications); add the implementation
* **encoder.py**: this is the base definition for the encoder, including the procedure to convert to a dictionary (see Encoding). Both encoding and decoding functions are now available that passes through an intermediatory dictionary translation.
* **encoders**: this folder hosts the definition of concrete instances of encoders (JSON for now).
* **message.py**: this is the definition of the OpenC2 message types (see Messages)
* **producer.py**: this is the entry point to send OpenC2 commands
* **profiles**: not sure this folder is really needed. It was originally conceived to host profile-specific definitions (like actions, targets, args, etc.)
* **response.py**: this file contains the data types for the response message
* **target.py**: this is the definition of the target object provided in the Language Specification (Sec. 3.3.1.2).
* targets.py: this is the list of available targets in the system. It should be populated with all the choices listed in Sec. 3.4.1, once their definition is available in datatypes.py.
* **transfer.py**: this is the base class for the transfer protocol. Currently it prints out the encoded message.
* **transfers**: the definitions of concrete transfer protocols. TODO: add the code for sending/receiving HTTP messages.

## Architecture

A white oval with black text

Description automatically generated

There are 2 entry points to use the library: producer and consumer. Every user that wants to use the OpenC2 library creates 1) a Producer to sending commands and receiving responses; 2) a Consumer to receive commands and sending responses.

Differently from what I originally thought, the actions involved by the OpenC2 message should be undertaken by an external component (the Security Function in the picture above). The library is agnostic of the concrete implementation of the Security Function. So for what concerns the thesis, the library must be limited to the components listed in the previous Section (including the slpf/nprobe profiles), and the concrete code to run iptables commands and nprobe must lay outside of the library.

For what concerns the classes, the following picture shows how they should be used to exchange OpenC2 messages.

A screenshot of a computer

Description automatically generated

## Actions

The Actions are now a simple enumeration of keywords, fully compliant with the standard. The original idea of associating a code to an action is not convincing because of the reasons discussed in Architecture. Additional actions envisioned by profiles should be added in the profile folder, using the static method provided by the Actions class.

## Messages

This is another major change to my previous understanding. After re-considering the code in a more thorough manner, I came to these conclusions:

1. The correct terminology used by the language specification is Command and Response, so we will use these from now on. Indeed, the transfer specification for HTTP might seem ambiguous on this point, since it refers to Request/Response. However, this terminology is referred to HTTP, not OpenC2: an HTTP Request carries an OpenC2 Command, and an HTTP Response carries an OpenC2 Response (same term in the second case).
2. Command/Response are a type of content, so they are now derived from this base class. Indeed, the class Content is empty, but we use it to have a common reference to hold both a Command and a Response element (this is used in the Message class).
3. Message is indeed poorly defined in the standard, and I really realized that only recently. The language specification lists the fields of the Message element at the beginning of Sec. 3.2, but does not dictates its structure as for the other elements. I came to the conclusion that there is not explicit definition of a Message structure (this intuition is confirmed by the fact the all examples in the language specification only show Commands/Responses, but not full Messages). Only after reading carefully the text, I noticed that “*transfer specifications define the on-the-wire format of a Message*”, which means that only the concrete transfer specification defines the full Message structure (e.g., HTTP in Sec. 3.3.2). The class Message is therefore now conceived to carry the metadata that will be used to create the Message, but their usage is left to the specific transfer protocols (see the current example for HTTP/HTTPS).

## Base types

Base types defines the types and structures defined by the Language Specification in Sec.3.1 and that defines the type of all message elements. Every base type must implement two methods: “todict” and “fromdict” (the latter must be a class method).

These two methods implement the code to translate an object instance to a dictionary and to build an object instance from a dictionary. These operations represent the intermediary dictionary translation described in the Encoding Section.

TODO: add the main rules and guidelines to write todict/fromdict methods for additional objects.

TODO: the Openc2Type definition is likely useful at this stage (it was used in a previous version. This could me removed in the following, after final check of its uselessness.

## Encoding

Encoding is perhaps the major change from the previous approach. I was not totally satisfied of the approach of adding a *tojson* (and in the future, *toyaml*, *toxml*, …) method to all basic elements, since this is not easily maintainable in time. However, I insisted with this approach because in case new elements are added, their conversion could be defined by their implementors, without changing the base Encoder class.

However, while playing with Python, I discovered that every object (=class) comes with a dictionary representation, where all field names and value are present. And the conversion of a dictionary to json is trivial with the *json* package. This suggested me the idea of using an intermediary representation of all messages as dictionary, and then translate this representation to json or any other format:

*Python objection à dictionary à json, xml, yaml, …*

So now a to\_json method for each object is no more necessary, because the conversion to dictionary is standard and can be done in a general way for any element. There is therefore a *todict* method in the base Encoding class, which can be called by the concrete encoder implementations to get the dictionary before encoding it.

The encoding/decoding operations happen on two layers. At the bottom layer, the Encoder defines the main rules to iteratively traverse complex object definitions and dictionaries. On the top layer, each OpenC2 type defines the rule to convert an instance to a dictionary and vice versa (todict, fromdict). The top layer relies on the bottom layer to recursively translate instances to a dictionary; the bottom layer relies in turn on the top layer to recursively create instances from a dictionary.

**Important note**. This approach works if every element perfectly matches the terminology and field order required by the language specification. Some constraints on fields order in a Record could be related, but it is better to keep the same order of the Language Specification to avoid misordering in the final encoded format.

The drawback of this approach is that any field that is not foreseen by the specification must be kept private, to easily remove it from the dictionary. The *todict* method indeed has a number of tricks to solve common issues with this approach (e.g., the *from* field that cannot be used because it is a Python keyword). In this case, the suggestion is to append an underscore, as already happens for the *from* field, which is easy to remove. There are also other issues that come out when combining multiple fields together (e.g., Action, Target, Args, Actuator in a Command, or IPv4Net in a Target). This is now solved for all base types currently defined, but additional rules must be added when the missing base types are added..

## Producer

Producer is the entry point abstraction to send Commands to a peer and receive its responses. It should kept all relevant data of the communication (e.g., name of the producer). However, each message can be sent to a different endpoint from the same producer (hence, the corresponding attributes should be passed to the *send* method).

TODO: add the function to receive the response (blocking/non-blocking?)

## Transfer

I partially implemented the HTTP transfer class to show how the whole payload could be created in a more elegant way from the Message metadata, and by exploiting the Encoding function.

TODO: add specific code to send/receive HTTP messages.

## External code

As mentioned at the beginning, the elements indicated as Security Controller and Security Functions are external to the Python library, so keep this code outside the *openc2* folder.

I provided an example in the main\_producer.py file to show how to use the Producer class. It must be extended with the code to receive and process responses.