Coding a Moon Cloud probe

The Moon Cloud team

1 Basic concepts

A probe is the core of the Moon Cloud assurance system, conceptually is a Python class that inherits from a set of already-provided classes (aka the Moon Cloud driver), it implements a specific control within a well-established structure.

The results of the probe will constitute a single Test Execution, the foundamental part of a Evaluation Execution (a set of Test Execution).

It takes a Python dictionary as input; the input contains all the parameters needed for its execution, including, but not limited to, the target of the test.

It produces three results as follows.

- `integer result`: a Integer field which indicates whether the probe did not find any discordance with the properties it has to assert and possibly the specific error happened, currently the options are `INTEGER_RESULT_TRUE`, `INTEGER_RESULT_FALSE`, `INTEGER_RESULT_TARGET_CONNECTION_ERROR`, `INTEGER_RESULT_TARGET_EXECUTION_ERROR`, `INTEGER_RESULT_INPUT_ERROR`, `INTEGER_RESULT_MOON_CLOUD_ERROR`. The latter is also returned when the probe encountered some errors during its execution.
- `pretty result`: a short string detailing why the probe finished with such result.
- `extradata`: an arbitrary set of values that contains the evidence the probe has collected. Consider a probe evaluating the proper setup of a TLS channel: it would return, for example, the list of weak ciphersuites, a set of recommendation... In general, `extradata` can contain any useful information for the final user.

The probe is packaged as a Docker container, making it self-contained. Using a container enables a smooth dependency management, since every tool, including external programs, is included in the container during the build.

In production the probe will be inserted into a chain of microservices, in particular the input will be injected from the Kubernetes Manager and the output will be passed along to the Evidence Writer. At the end the user will be presented with the results of the evaluation.

2 Finite state machine

A probe is a **finite state machine** with two so-called *chains* (flows), as follows.

- The main chain (or *forward* chain) which is the one that is usually executed unless there is an error.

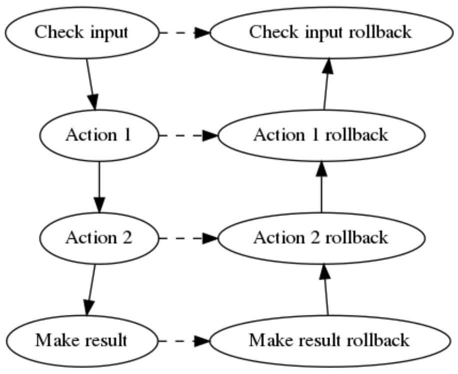

Figure 1: State machine

- The backward chain (or *rollback* chain) which is only called in case of an error.

Each chain is made up of several states that concretely are Python methods, and each state represents a *step* of the control.

Typically there are at least three steps (and as such three states and three methods).

1. The *initial phase* receiving the input and checking its correctness. Also, it may setup some attributes for the next steps.
2. A step executing the actual action, such as invoking an external tool.
3. A step evaluating the previous results and fixing `result`, `pretty result` and `extradata`.

There exists a correspondence between a state in the main chain and one in the rollback chain, where the state in rollback chain must *rollback* the actions executed by the correspondent state in the forward chain in case of an error.

The actions that have to be rollback-ed are actions that somehow *modify* the target, otherwise the rollback *state* can be just empty.

3 Standard project files

This section describes other files, not strictly related to coding, that are needed to have a fully-working probe.

3.1 Dockerfile

The `Dockerfile` is needed to build the probe. Every image must use as base image **`python:3.10 (example version)`**, which is an image based on Debian. Every tool that is needed must be installed manually. In Build and run, starting from page 12, we describe rules for building the image.

```
FROM python:3.10

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app
ADD requirements.txt /usr/src/app
RUN pip install --no-cache-dir -r requirements.txt
ADD . /usr/src/app
```

```

ENTRYPOINT ["/bin/bash", "-c"]
CMD ["/usr/src/app/probe/probe.py"]
```

3.2 requirements.txt

The requirements file contains all the pip's requirements. It must contain the driver reference, the version must be the same as the current production driver.

```
git+https://repository.v2.moon-cloud.eu/dev/driver.git@v1.0.0
```

3.3 readme.md

The readme **must** contain the following information.

- A short description of what the probe does.
- Eventual external tools being used.
- A sample of the input. Use comments (in json5 they are allowed) to clarify non-standard fields and any other useful information.
- A sample of the output, for which the same rules of the input are applied.

You also need to clarify if you depend on something installed on the target. For example, if you need SSH enabled on the target, state this requirement in the `readme.md`. If you need more exotic dependencies, add a *state* in your code that checks if they are installed as soon as possible, before doing other actions. If those dependencies are not present, return a meaningful message to the user through `extradata`.

Example:

~~~
# probe-name

This probe is just a sample probe. It uses [nmap](https://nmap.org/) to scan for open ports on the
↪ target system.

## Input

```json5
{
    "config": {
        "host": "unimi.it"
    }
}
```

## Output

```json5
{
    integer_result: 0,
    pretty_result: "Test executed successfully",
    extra_data: {
        "Vulnerabilities": 0,
        "Remediation": {
            "1": {
                // and so on
            }
        }
    }
~~~

~~~~
}
```
~~~~

3.4 .gitignore

No IDE-related files are allowed.

```
# for python virtual environment
env/
venv/

# python cache
__pycache__/
*.pyc

# ide-related stuff
.idea/
.vscode/

# Moon Cloud credential test file
credential.json

```

3.5 .dockerignore

```
# for python virtual environment
env/
venv/

# python cache
__pycache__/
*.pyc

# ide-related stuff
.idea/
.vscode/

# also add git-related files
.git/

# also the readme
readme.md

# and also the Moon Cloud credential test file
credential.json

```

4 Coding a probe

4.1 Project organization

The probe is made up of several files, most of them are always the same for every probe.

A good project structure looks like the following.

```

probe/input.json
probe/probe.py
```

```
probe/schema.json
probe/test.json

# git and gitlab-related files
.gitignore
.gitlab-ci.yml

# docker-related files
Dockerfile
.dockerignore

# dependencies
requirements.txt

# documentation
readme.md
```

- The `probe` directory is where the specific code of each probe goes.
- `.gitignore` must exclude basically everything that is different from the structure shown, including IDE-related files and virtual environment.
- `.gitlab-ci.yml` is the file enabling the CI/CD of the probe.
- `Dockerfile` – more on this in section 3.
- `.dockerignore` is at least as restrictive as `git`'s, but you have to include also `.git/` and `readme.md`.
- `requirements.txt` holds `pip`'s requirements, it always contains the driver reference (details in section 3).
- `readme.md` holds project documentation – more on this later.

The actual code to write usually consists of (at least) one Python file under the `probe` directory, you can name it whatever you like, but it's better to use `probe.py` as a name.

- `input.json` will be populated in production with the run-time input of the Probe.
- `schema.json` contains the scheme that will be used to validate the input of the Probe.
- `test.json` contains an example of the Probe input.

5 Coding

The probe is effectively a class that inherits from `AbstractProbe`, which is located into `abstract_probe.py`.

The input of the probe is a Python dictionary that can be accessed by using `self.config.input`. The keys of the dictionary can be accessed by using the common Python syntax for dictionary-lookup. Instead of using `["property"]`, you should instead use the `get("property")` method.

In the remainder of this section we will discuss the logic steps that made up a probe.

5.1 Dealing with the input

The most common parameter for a probe is the target ip address, or url. We have, indeed, two types of targets `url` or `host` [1]

[1]There exists more host types, but they are not common.

A very simple input looks like this:

```
{
    "config": {
        "url": "https://www.google.com"
    }
}
```

It is important to remember that parameters such as ip address, hostname, url, port are always kept under the `config` key.

If you have more parameters, you can set them under another object:

```
{
    "config": {
        "url": "https://www.google.com"
    },
    "Scan settings": {
        "threshold": 100,
        "boost": 20
    }
}
```

Usually all the input-related stuff is placed into one method called `parse_input` or something similar. By using the `get()` method on `self.config.input` you get a possible `None` value, if you don't specify a default. When dealing with input, keep in mind the following rules.

- Assume a default value for each parameter whenever is possible. For example, 443 is a good default for a port number, in a probe targeting an HTTPS-enabled website.
- The input is validated with the schema.json file, so you can assume a valid input when the probe starts.

The following code is a good example.

```
# probe/probe.py

class MyProbe(abstract_probe.AbstractProbe):

    scanner = None

    def parse_input(self, inputs=None):
        config = self.config.input.get('config')

        # no default value on the host - of course
        self.target = config.get('host')

        # instead, we can rely on 80 as a good default for a web-server
        # targeting probe
        self.port = config.get('port', 80)

        scan_settings = self.config.input.get('scan_settings')
        min_threshold = scan_settings.get('minimum_threshold', 10)
        boost = scan_settings.get('boost', 20)

        # finally create the scanner
        self.scanner = Scanner(self.target, self.port,
            self.min_threshold, self.books)
```

In `test.json` you should provide a sample input that can be used for testing the probe.

We don't have many guidelines to structure the input, but the following.

- In the section `config` place only keys related to connect to the target. For ports, assume the protocol standard as default, such as 22 for SSH, 80 for HTTP, 443 for HTTPS.
- Use other top-level keys for test configurations, such a minimum threshold and so on; assume a reasonable default.

In `schema.json` you should provide a valid jsonschema in accordance to the expected input of the probe. For example:

```
{
    "type": "object",
    "properties": {
        "config": {
            "type": "object",
            "properties":{
                "host":{
                    "type":"string"
                }
            }
        }
    }
}
```

5.1.1 Using Credentials

If your probe needs to use credentials to connect to the target, you can specify your requirement by defining the method `requires_credential` as follows:

```
    def requires_credential(self):
        return True
```

After that you can assume that the required credential will be available in the `self.config.credential` variable. In the development phase, for testing purposes only, you can create a file named `credential.json` in which you can put the credential to inject. Please remember to include it in the `.gitignore` file before pushing to the repo.

The following example shows how we can specify a set of SSH credentials required to connect to an host. The possible inputs you have to deal with are: *i*) username and password, *ii*) username and private key, *iii*) username, private key and private key passphrase. These parameters will be inside the file `credential.json`. The other configs(host, port, etc.) will be present in the file `test.json`.

Username and password:

```
{
    "username": "you",
    "password": "qwerty"
}
```

Username and private key.

```
{
    "username": "you",
    "private_key": "---BEGIN SSH private key ---\njnwi35n....ih\n --- END SSH private key ---"
}
```

And finally, username, private key and private key passphrase to decrypt an encrypted private key.

```
{
    "username": "you",
    "private_key": "---BEGIN SSH private key ---\njnwi35n....ih\n --- END SSH private key ---",
    "private_key_passphrase": "pass1234"
}
```

The code will look like the following.

```python
import abstract_probe

class Probe(abstract_probe.AbstractProbe):

    def check_input(self, inputs=None):
        # basic input
        config = self.config.input.get('config')
        host = config.get('host')
        port = config.get('port', 22)

        username = self.config.credential.get('username')

        password = self.config.credential.get('password')
        private_key = self.config.credential.get('private_key')
        private_key_passphrase = self.config.credential.get('private_key_passphrase')
        # verify there is a valid sets of parameters
        assert password is not None or private_key is not None or (private_key is not
            None and private_key_passphrase is not None)
        # verify there are not useless combinations, such as passphrase but
        # no private key
        assert (private_key_passphrase is not None and private_key is None) and
            (password is not None and private_key is not None) and
            (password is not None and private_key_passphrase is not None)

    def requires_credential(self):
        return True

```

A class implementing SSH connection will be provided by the team.

5.2 Dealing with the output

The output is a Python dictionary that gets serialized into JSON. Let's imagine the probe execute a series of small tests, how the output should be organized? There are basically two ways.

The first is the following.

```
// a list in which each item is the output of the single sub-test
// assuming the the object structure is defined by the probe itself
{
    "integer_result": 0,
    "pretty_result": "The probe executed successfully!",
    "extra_data": [
        {
            "TestName":"Test1",
            "Passed": true,
            "Score": 10,
            "Description": "..."
        },
        {
            "TestName":"Test2",
            "Passed": false,
            "Score": 1,
            "Description": "..."
        }
    ]
}
```

The second and the preferred one is instead to have a very *nested* dictionary, where which nesting level represents a lower level of detail.

A very simple example is the following.

```
{
    "integer_result": 0,
    "pretty_result": "The probe executed successfully!",
    "extra_data": {
        "Summary": "1/2 succeeded",
        "Tests": {
            "Test1": {
                "Score": 10,
                "Description": "...",
                "Mitigations": {
                    "Mitigation1": {
                        "Description": "...",
                        "Suggested": true
                    },
                    "Mitigation2": {
                        "Description": "...",
                        "Suggested": false
                    }
                }
            }
        }
    }
}
```

Adding a top-level key to the result is as easy as calling the method put_extra_data() `self.result.put_extra_data(key, value)`, see the example below.

```
def action1(self, inputs=None):
    results = execute_action_1()
    self.result.put_extra_data("TOP_LEVEL_KEY_1",{"Score": results["score"],
        "Info": results["info"]})

 def action2(self, inputs=None):
    results = execute_action_2()
    self.result.put_extra_data("TOP_LEVEL_KEY_2",{"Score": results["score"],
        "Info": results["info"]})
```

Will produce the following.

```
{
    "integer_result": 0,
    "pretty_result": "This test...",
    "extra_data": {
        "TOP_LEVEL_KEY_1":{
            "Score": 10,
            "Info": "This test is..."
        },
        "TOP_LEVEL_KEY_2":{
            "Score": 1,
            "Info": "The result is..."
        }
    }
}
```

Also, by looking at the snipped above, you may have noted that the `self.result.put_extra_data(key, value)` can be called multiple times, in fact every time that call is issued another top-level key is added to the result. Moreover, you can use such method in every part of the code, you are not forced to use it in the last *states* (though you will probably end up doing it).

5.3 Dealing with the probe structure

You are free to create all the methods and classes you need, and not all the methods of the class inheriting from `abstract_probe.AbstractProbe` automatically become a *state*.

States will just be the top-level methods representing an action in the probe execution, but you can decide which is the correct level of abstraction. You can think of *states* as the public methods you whish to expose to others.

Then, you need to *configure* the probe. You **must define** a method called **`atoms(self)`**, in which you register all the pairs of forward and rollback *states* by passing those pairs to the constructor of `AtomPairWithException`. You don't need to implement the class `AtomPairWithException`, it has already been implemented in the atom.py file.

```
def atoms(self) -> typing.Sequence[atom.AtomPairWithException]:
        return [
            atom.AtomPairWithException(forward=self.test1, forward_captured_exceptions=[
                # Assertion exception
                atom.PunctualExceptionInformationForward(
                    exception_class=AssertionError,
                    action=atom.OnExceptionActionForward.ROLLBACK,
                    # Specify result callback to merge with current result
                    result_producer=self.assertion_exc_callback
                )
            ], rollback=self.rollback_test1, rollback_captured_exceptions=[
                 # Assertion exception
                 atom.PunctualExceptionInformationRollback(
                    exception_class=AssertionError,
                    action=atom.OnExceptionActionForward.STOP,
                    # Specify result callback to merge with current result
                    result_producer=self.assertion_exc_callback
                )
            ]),
            atom.AtomPairWithException(forward=self.test2)
        ]
```

`AtomPairWithException` fields in detail:

- `forward`: a function reference that specify the step to be executed
- `forward_captured_exceptions`: a list of the exceptions that have to be captured
- `rollback`: a function reference that specify the rollback step to be executed
- `rollback_captured_exceptions`: same as `forward_captured_exceptions` for rollback

To specify a `PunctualExceptionInformationForward/PunctualExceptionInformationRollback` you need to define the following fields:

- `exception_class`: the exception class to capture
- `action`: the action to do when the exception is raised, the available options are:
  - `GO_ON`: proceed on forward/rollback (default)
  - `ROLLBACK`: move on the rollback flow (available only in `PunctualExceptionInformationForward`)

    - `STOP`: stop the execution (no additional states are executed)

- `result_producer`: an optional callback to merge/modify the result after the exception occured, it takes as input the captured exception and returns a `Result`

More documentation on the driver syntax will be provided by the team.

5.4 Dealing with rollback

Each method that corresponding to a state of the FSM has a related *rollback method*. Such method is responsible of *undoing* the operation performed by the forward one on the target.

Most of the time, you don't need to undo an action and you can just write an empty rollback. For example if you just issue an `HTTP GET`, or you just list the list of ciphersuites offered on HTTPS, you don't need to undo anything. Instead, you need to undo an operation like creating a directory. Note that in this very specific case you can also create a temporary directory.

Attention: rollback functions are only called when **an error occurs** on the forward chain. If you need to clean up the target after having run some action, just place another action in the bottom of the forward chain.

So when do errors occur? An error is every **uncaught exception in the forward chain**, that immediately stops the forward chain and starts the corresponding rollback one. As such, you don't need to catch every possible exception, just catch the ones that are meaningful for the probe output. As an example, consider a probe that issues an HTTP request against a target. It can cause an I/O error; this is an exception that you are interested in, because it can signal that the host is not up.

```
atom.AtomPairWithException(forward=self.test1, forward_captured_exceptions=[
                # Assertion exception
                atom.PunctualExceptionInformationForward(
                    exception_class=AssertionError,
                    action=atom.OnExceptionActionForward.ROLLBACK,
                    # Specify result callback to merge with current result
                    result_producer=self.assertion_exc_callback
                )], rollback=self.rollback_test1, rollback_captured_exceptions=[
                # Assertion exception
                atom.PunctualExceptionInformationRollback(
                    exception_class=AssertionError,
                    action=atom.OnExceptionActionForward.STOP,
                    # Specify result callback to merge with current result
                    result_producer=self.assertion_exc_callback
                )]
)
```

For example, assertions are typically used in the method that checks a condition. When an assertion is not satisfied an exception is thrown and the execution is stopped(if the associated action is STOP).

5.5 Dealing with passing data through methods

A *state* often needs to make some computed values available to the methods in the chain. To achieve this there are basically two ways.

1. Define class attributes that will hold those values (please initialize attributes in the constructor even with an empty value, to make the code clear)
2. *Pass* the variables to the next method.

The first one is self-explanatory. Related to the second, every *state* can return values, those values will be received by the next method by its `input` parameter. This is why each *state* has a fixed signature of `(self, inputs)`. `inputs` especially can be a variable of every type, and will contain what the previous method has returned. The `Atom` class will take care of this.

It's up to you to decide which way is better, as a rule of thumb consider that values that are needed in every *state* (or most of them) can become class attribute, otherwise you can use the second strategy.

If you like to follow the second one, just make a heavy of assertion in the receiver method, to make sure the data you are receiving are correct.

An example.

```python
def action_1(self, inputs=None):
    result = make_http('https://example.com', 'post')
    return json.loads(result.body)

def action_2(self, inputs):
    body = inputs
    assert body is not None
    # make also other assertions if you need, to make sure
    # `inputs` is as you except
```

5.6 Dealing with the result

It is important to handle this properly since the result is what marks an evaluation as *successful* or not. In the previous sub-section we have seen how methods can return values and how they are consumed. In fact, the final probe result is just the result of the last method registered within `self.AtomPairWithException`.

The value `INTEGER_RESULT_TRUE` is especially returned only when everything was successful. What *successful* means actually depending on the specific probe. Generally speaking, when you are writing a probe that checks the compliance against a standard/recommendation only return `INTEGER_RESULT_TRUE` if the compliance is confirmed.

In case the rollback chain is called, the result will automatically be `INTEGER_RESULT_MOON_CLOUD_ERROR` or a more specific error that you have defined.

6 Build and run

6.1 Prepare the dependencies

Before you can running the code or building the image you need to download the Moon Cloud driver, please ensure that the requirements.txt file contains the reference to the driver. Install the driver with:

```
    pip install -r requirements.txt
```

You'll be prompted to insert your git credentials(the same you use on the GitLab repository), after that you should be have downloaded correctly the driver(and the other dependencies if you specified).

If you need to build the probe locally you'll need to create a personal Access Token, to do that login to your GitLab account, go to your profile settings, click on Access Tokens and generate a new Access Token as follows:

- Insert a token name as you like (this will be the GITLAB_TOKEN_USER)
- Select an adequate expiration date

- Check only the scope read_repository
- Click on generate a new token

You should now see the generated token (this will be the GITLAB_TOKEN), please write it down immediately as it will not be saved on GitLab.

Note that when you push to the GitLab registry you don't need to specify a token, the CI/CD process will handle it for you.

6.2 Build the probe

To build the image of the probe run the proper Docker command as follows.

```
docker build -t <image-name> --build-arg GITLAB_TOKEN_USER=${GITLAB_TOKEN_USER}$ --build-arg
↪  GITLAB_TOKEN=${GITLAB_TOKEN} .
```

Any external dependency must be added before adding the code itself, to let *Docker* use its cache system. If you need some build-time dependencies, such as a compiler, remove them before adding your code, since they are not needed for the execution but just for the build. You can eventually use *multi-stage builds*.

```
FROM python:3.10

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

# add dependencies here
# use apt update && apt install always in the same RUN
# directive
RUN apt update && apt install -y make gcc git
RUN git clone --depth 1 https://github.com/your/dependency.git && cd dependency && \
    ./configure && \
    make -j && \
    make install -j
# now remove build-time dependencies. If you used `git` to download
# the tool, you can then remove it
RUN apt autoremove -y gcc make git

# now add your code

ADD requirements.txt /usr/src/app
RUN pip install --no-cache-dir -r requirements.txt
ADD . /usr/src/app

ENTRYPOINT ["/bin/bash", "-c"]
CMD ["/usr/src/app/probe/probe.py"]
```

6.3 Run the probe

To test the probe locally, use the sample input provided in `probe/test.json`. Make sure it is a legitimate operation, for example, don't try SQL Injection on a random website. The input is piped to the probe `stdin`, as such you need to do the following.

```
    cat probe/test.json | python probe/probe.py
```

Or if you also want to test with the local builded Docker image:

```
cat probe/test.json | docker run -i <image-name> "python probe/probe.py"
```

In general, to pass any input you can also use the `echo` command.

```
echo '{"config":{"host":"192.168.100.2"}}' | docker run -i <image-name> "python probe/probe.py"
```

Note:
Please make sure the method `entrypoint.start_execution()` has been invoked at the end of the `probe.py` file as follows:

```
if __name__ == '__main__':
    entrypoint.start_execution(Probe)
```

7 Rules

When you code keep in mind the following rules.

Code-related aspects Simple is better, explicit is better than implicit. Always document your code, make it clear what it's going on, especially when it's not obvious. Newer versions of Python support annotating functions and variables with their type: do that, especially on complex code, but also everywhere you think is useful. They helps your IDE in catching more errors. Follow established conventions for writing code, such as *snake case* for naming variables and methods, *camel case* for naming classes, using contexts when possible, and so on.

Probe-related aspects Choose the right level of abstraction when registering methods within `atoms`. Make sure the last method registered in `atoms` returns either `INTEGER_RESULT_TRUE` or `INTEGER_RESULT_FALSE`. Follow the rules stated here for organizing the probe code, also, you can split the code into multiple files or even directories if they get too big and unmanageable to you; just place everything under the `probe` directory. Especially, follow our guidelines for structuring inputs and outputs.

***Docker* packaging** Use the best practices described in section 6, such as adding dependencies before adding the actual code. If you rely on an external tool, make sure your Dockerfile use a fixed version that is known to work. Don't do `RUN wget -O tool.zip https://tool.exampke.com/latest.zip`; instead: `git clone https://github.com/example/tool.git` and `git checkout tags/v2.4` where `2.4.7` is a recent version that you know works fine. Finally, remove build-time dependencies, in this case you will remove `git`. You may also statically include the dependencies into the project.

Other aspects Use `PyCharm` as your IDE, trust it and fix the warnings it suggest you. You will use `git` to manage your projects, follow the best practices for working with it, such as providing meaningful messages where you commit.

8 Example

A full working example is presented. It is the probe checking for the avaiability of a resource through *Curl*.

```
#!/usr/bin/env python
import copy, typing, subprocess
from mooncloud_driver import abstract_probe,atom,result,entrypoint

# Curl connection test
# Trying curl connection, content request and JSON conversion
class Probe(abstract_probe.AbstractProbe):
    CURL_CMD = "curl -sS {host}"

    # NOTE: this structure is only an example, feel free to organize to
    # output structure as you like
    RESULT_MAP = {
        result.INTEGER_RESULT_TRUE: {
            'pretty_result': "Curl test executed succesfully",
            'base_extra_data': {}
        },
        result.INTEGER_RESULT_FALSE: {
            'pretty_result': "Error occured during the test execution",
            'base_extra_data': {
                'Error': "{e}"
            }
        },
        result.INTEGER_RESULT_TARGET_CONNECTION_ERROR: {
            'pretty_result': "Target connection error",
            'base_extra_data': {
                'Error': "{e}"
            }
        },
        result.INTEGER_RESULT_INPUT_ERROR: {
            'pretty_result': "Input error",
            'base_extra_data': {
                'Error': "{e}"
            }
        },
        result.INTEGER_RESULT_MOON_CLOUD_ERROR: {
            'pretty_result': "Assertion failed",
            'base_extra_data': {
                'Error': "{e}"
            }
        }
    }

    sb_returncode: int = -1
    sb_stderr: str = ""

    def test1(self, inputs: any) -> bool:
        # Getting host from probe input and make request
        host = self.config.input['config']['host']
        # If find a 'custom_target' key ignore the 'host' key
        if 'custom_target' in self.config.input['config']:
            host = self.config.input['config']['custom_target']
        assert host is not None and host != "", "host input field not valid"
        sb = subprocess.run(self.CURL_CMD.format(host=host).split(" "), stdout=subprocess.DEVNULL,
        ↪  stderr=subprocess.PIPE)
        # Getting the return code and output
        self.sb_returncode = sb.returncode
        assert isinstance(self.sb_returncode, int), f"Inputs expected int but
        ↪  got{type(self.sb_returncode)}"
        if sb.returncode != 0:
            self.sb_stderr = str(sb.stderr)
        return True

    def test2(self, inputs: bool) -> bool:
        # Converting return code to proper values
        converted_return_code: int = 0
        if self.sb_returncode == 1 or self.sb_returncode == 3:
            converted_return_code = 4
        elif self.sb_returncode == 6 or self.sb_returncode == 7 or self.sb_returncode == 28:
            converted_return_code = 2
        elif self.sb_returncode != 0:
            converted_return_code = 1
        # Setting result properties
        result_obj = self.RESULT_MAP.get(converted_return_code)
        self.result.integer_result = converted_return_code
```

```python
        self.result.pretty_result = result_obj['pretty_result']
        base_extra_data = copy.deepcopy(result_obj['base_extra_data'])
        if converted_return_code != 0:
            base_extra_data['Error'] = base_extra_data['Error'].format(e=self.sb_stderr)
        self.result.base_extra_data = base_extra_data
        return True

    def atoms(self) -> typing.Sequence[atom.AtomPairWithException]:
        return [
            atom.AtomPairWithException(forward=self.test1, forward_captured_exceptions=[
                # Assertion exception
                atom.PunctualExceptionInformationForward(
                    exception_class=AssertionError,
                    action=atom.OnExceptionActionForward.STOP,
                    # Specify result callback to merge with current result
                    result_producer=self.assertion_exc_callback
                )
            ]
                                           ),
            atom.AtomPairWithException(forward=self.test2)
        ]

    def assertion_exc_callback(self, e):
        base_extra_data = copy.deepcopy(self.RESULT_MAP.get(5)['base_extra_data'])
        base_extra_data['Error'] = base_extra_data['Error'].format(e=e)
        return result.Result(
            integer_result=5,
            pretty_result=self.RESULT_MAP.get(5)['pretty_result'],
            base_extra_data=base_extra_data
        )

if __name__ == '__main__':
    entrypoint.start_execution(Probe)
```

A simplified version of the `Dockerfile` is shown below. It installs *Curl* to provide it to the Probe.

```dockerfile
FROM python:3.9

ARG GITLAB_TOKEN_USER
ARG GITLAB_TOKEN

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app
ADD requirements.txt /usr/src/app

RUN sed -i -e 's/git+https:\/\/repository.v2.moon-cloud.eu\/dev\/driver.git/git+https:\/\/${GITLAB_
↪  TOKEN_USER}:${GITLAB_TOKEN}@repository.v2.moon-cloud.eu\/dev\/driver.git/'
↪  requirements.txt

RUN pip install --no-cache-dir -r requirements.txt
ADD . /usr/src/app

RUN apt-get update && apt-get install curl -y

ENTRYPOINT ["/bin/bash", "-c"]
CMD ["/usr/src/app/probe/probe.py"]
```

Finally, the file `readme.md`

```markdown
# Curl test probe

Testing probe using curl
```

~~~

Dependencies

- Curl

Input

> :warning: Note: if find a **custom_target** key it will ignore the **host** key

```json5
{
    config:{
        host:"test.com"
    }
}
```

Output

```json5
{
  integer_result: 0,
  pretty_result: "Curl test executed successfully",
  extra_data: {
    //Errors and details
  }
}
```
~~~

9 Contacts

For questions reach us at:

- Nicola Bena nicola.bena@unimi.it
- Antongiacomo Polimeno antongiacomo.polimeno@unimi.it
- Alex Della Bruna alex.dellabruna@studenti.unimi.it