Chaos Monkey Engine Documentation

Release 0.1

BBVA Innotech

Oct 25, 2017

Contents:

Chaos Monkey Engine is a tool designed to perform attacks against any infrastructure. The engine gives you a couple of attacks and planners out of the box, and offers a simple interface to create your own.

The engine uses Apache Libcloud to access cloud providers and the AWS driver is implemented out of the box (check the *Extending the Chaos Mokey Engine* section if you want to create your own driver).

The Chaos Monkey Engine also exposes an API to manage attacks and planners.

If you are in a hurry, try the *Quickstart* to see it in action. If you want to know more, jump to *Chaos Monkey Engine Overview*.

CHAPTER 1

Quickstart

This section will show how to deploy an initial Chaos Monkey Engine in a few minutes to test its functionality.

Chaos Monkey Engine

Chaos Monkey Engine helps you planning and executing attacks against any infrastructure. This helps you detecting possible improvements in the mission of building an antifragile infrastructure.

Requirements

- Docker 1.12+
- Amazon Web Services credentials in environment variables:
  - `AWS_ACCESS_KEY_ID`
  - `AWS_SECRET_ACCESS_KEY`
  - `AWS_DEFAULT_REGION`

Get your engine up and running

Build and run the Docker container:

```
docker build -t chaos-monkey-engine .
docker run --rm -p 5000:5000 -e AWS_ACCESS_KEY_ID -e AWS_SECRET_ACCESS_KEY -e AWS_
↪DEFAULT_REGION -ti chaos-monkey-engine
```

The Chaos Monkey Engine should be now listening in port 5000 TCP and ready to attack the machines in your AWS infrastructure.

Schedule a simple attack

Create a plan file (`plan.json`) with a content similar to this one:

```
{
    "name": "Terminate random running instance",
    "attack": {
        "ref": "terminate_ec2_instance:TerminateEC2Instance",
        "args": {
            "filters": {
                "instance-state-name":"running"
            }
        }
    },
    "planner": {
        "args": {
            "min_time": "10:00",
            "max_time": "18:00",
            "times": 3
        },
    "ref": "simple_planner:SimplePlanner"
    }
}
```

This plan schedules 3 attacks between 10:00 and 18:00 that terminate running EC2 instances of the region selected with `AWS_DEFAULT_REGION`. You can use filters as described in the official AWS documentation.

Send the plan to the engine:

```
curl -X POST -H "Content-Type:application/json" -d @plan.json localhost:5000/api/1/
↪plans/
```

Once the plan has been executed and the attack executors are created, you can check them issuing the following request:

```
curl localhost:5000/api/1/executors/
```

Monitoring the output of the Chaos Monkey Engine, you will see the resulting executions.

CHAPTER 2

Installation

There are two installation methods: Docker container (**recommended**) and Python package.

Docker container

Build the container with Docker 1.12+:

```
docker build -t chaos-monkey-engine .
```

Then, you can run the container as the `chaos-monkey-engine` command. E.g:

```
docker run --rm chaos-monkey-engine --help
```

Python package

You can install the latest package from PyPi:

```
pip install chaosmonkey
```

Now you have the `chaos-monkey-engine` command on your path:

```
chaos-monkey-engine --help
```

CHAPTER 3

Usage

Running `chaos-monkey-engine --help` shows the usage information:

```
Usage: chaos-monkey-engine [OPTIONS]

  Chaos Monkey Engine command line utility

Options:
  -p, --port INTEGER          Port used to expose the CM API
  -t, --timezone TEXT         Timezone to configure the scheduler
  -d, --database-uri TEXT     SQLAlchemy database uri  [required]
  -a, --attacks-folder TEXT   Path to the folder where the attacks are stored
                              [required]
  -p, --planners-folder TEXT  Path to the folder where the planners are stored
                              [required]
  --help                      Show this message and exit
```

- The **port** defaults to 5000
- The **timezone** defaults to `Europe/Madrid`. The engine uses pytz for managing the timezones.

The Docker container has a default `CMD` directive that sets these sane default options:

```
"-d /opt/chaosmonkey/src/storage/cme.sqlite -a /opt/chaosmonkey/src/attacks -p /opt/
↪chaosmonkey/src/planners"
```

For example, to launch the server on port 5000 TCP as a foreground process, passing AWS credentials:

```
docker run --rm -p 5000:5000 -e AWS_ACCESS_KEY_ID -e AWS_SECRET_ACCESS_KEY -e AWS_
↪DEFAULT_REGION -ti chaos-monkey-engine
```

CHAPTER 4

Chaos Monkey Engine Overview

The Chaos Monkey Engine (CME) is a tool to orchestrate attacks to your cloud infrastructure in order to implement the principles of Chaos Engineering. It is inspired in the Netflix's SimianArmy but built with these principles in mind:

- Multi-cloud (not only AWS) support through standards as Apache Libcloud and SSH
- Ease of extensibility to add your new attacks and planners
- HAL API interface

The CME is completely API-driven, so that it can be easily integrated with external and third-party systems.

Implementation

The CME is a Flask application running with gevent. It uses the apscheduler engine to schedule attacks and SQLAlchemy to persist the state of the attacks.

Arquitecture

The API has two main resources, planners and attacks. Trough the API you use planners to schedule jobs (named *executors*) that execute attacks. You can build your own planners and attacks to fit your needs.

In the other hand, in order to implement the attacks, you will need to interact with the cloud providers. Although it is not exactly required, we recommend using apache-libcloud (which is included as a dependency) in order to build reusable attacks abstracted from the underlying provider, using their cloud compute drivers. A working example of driver factory for EC2 is included in the CME package in order to interact with AWS EC2 instances.

Planners and Attacks

Planners and attacks are the main resources of the engine. You create executors (scheduled attacks) using a planner and an attack definition.

The engine provides certain planners and attacks out of the box:

Planners

Exact Planner

Reference: `exact_planner:ExactPlanner`

A planner that schedules an executor for a specific date.

Example:

```
"planner": {
  "ref":"exact_planner:ExactPlanner",
  "args": {
    "date":"2016-06-21T15:30:12+02:00"
  }
}
```

Simple Planner

Reference: `simple_planner:SimplePlanner`

A planner that schedules N executors for specific time range in today.

Example:

```
"planner": {
  "ref":"simple_planner:SimplePlanner",
  "args": {
    "min_time" : "10:00",
    "max_time" : "18:00",
    "times": 4
  }
}
```

Attacks

Terminate EC2 instance

Reference: `terminate_ec2_instance:TerminateEC2Instance`

Issues a terminate on a random EC2 instance filtered by any AWS EC2 filter.

Example:

```
"attack":{
  "ref": "terminate_ec2_instance:TerminateEC2Instance",
  "args":{
    "filters": {"tag:Chaos":"true"}
  }
}
```

Terminate EC2 instance not excluded

Reference: `terminate_ec2_instance:TerminateEC2InstanceNotExcluded`

Issues a terminate on a random EC2 instance filtered and excluding instances with AWS filters.

Example:

```
"attack":{
  "ref": "terminate_ec2_instance:TerminateEC2InstanceNotExcluded",
  "args":{
    "filters": {"tag:Chaos":"true"},
    "excluded": {"availability-zone":"eu-west-1"}
  }
}
```

Api Request

Reference: `api_request.ApiRequest`

Makes a request to any API endpoint.

Example:

```
"attack":{
  "ref": "api_request.ApiRequest",
  "args": {
    "endpoint": "http://localhost:4500",
    "method": "GET",
    "payload": {"test": "1"},
    "headers": {"X-CUSTOM-HEADER": "test"},
  }
}
```

Run script

Reference: `run_script:RunScript`

Runs a script on a random EC2 instance filtered by any AWS instance tag. The instance must be reachable by SSH.

Example:

```
"attack":{
  "ref": "run_script:RunScript",
  "args": {
    "filters": {
      "tag:Chaos":"true"
    },
    "local_script": "script_attacks/s_burncpu.sh",
    "remote_script": "/chaos/burn_cpu",
    "ssh" : {
       "user": "ec2-user",
       "pem": "BASE64ENCODEDPEM"
    },
    "region": "eu-west-1"
  }
}
```

The local script is uploaded to the `remote_script` destination and executed. The *pem* for the credentials is the Base64 encoded version of the file.

CHAPTER 5

Extending the Chaos Mokey Engine

Adding Custom Planners

You can create your own planners to fit your needs. A planner is just a python class that receives some properties and and attack configuration. Based on this properties the planner schedule executors that execute attacks.

To create your custom planner, just create your module in the planners folder, and add a class that implements the Planner interface (*chaosmonkey.planners.planner()*).

Internally the engine uses apscheduler to schedule jobs (*executors*) to be executed sometime in the future.

Your planner class must have three mandatory properties (ref, schema and example) and two mandatory methods (`plan` and `to_dict`):

```
from chaosmonkey.planners import Planner

class MyPlanner(Planner):

    ref = "my_planner:MyPlanner"
    schema = {
        "type": "object",
        "properties": {
            "ref": {"type": "string"},
            "args": {
                "type": "object",
                "properties": {
                    "date": {"type": "string"}
                }
            }
        }
    }
    example = {
        "ref": "my_planner:MyPlanner",
        "args": {
            "date": "2016-06-21T15:30"
        }
```

```
    }

    def plan(self, planner_config, attack_config):
            """
            planner_config and attack_config are json/dicts passed to the endpoint
            when executing this planner.

            The plan method must call self._add_job to schedule attacks in a date.
            This date should be calculated based on planner_config variables.
            The executor config is passed as a parameter to the add_job, and it will
            be passed to the attack executor when the scheduler executes a job to␣
↪execute
            the attack.
            """
            self._add_job(attack_date, "Job Name", attack_config)

    @staticmethod
    def to_dict():
        return Planner._to_dict(MyPlanner.ref, MyPlanner.schema, MyPlanner.example)
```

Adding Custom Attacks

You can create your own attacks. An attack is a Python class that receive some properties and execute an attacks based on the properties.

To create your custom attack, just create your module in the attacks folder, and add a class that implements the Attack interface (*chaosmonkey.attacks.attack()*)

Your attack class must have three mandatory properties (ref, schema and example) and two mandatory methods (run and to_dict):

```
from chaosmonkey.attacks import Attack

class MyAttack(Attack):

    ref = "my_attack:MyAttack"

    schema = {
        "type": "object",
        "properties": {
            "ref": {"type": "string"},
            "name": {"type": "string"}
        }
    }

    example = {
        "ref": "my_attack:MyAttack",
        "name": "attack1"
    }

    def __init__(self, attack_config):
        super(MyAttack, self).__init__(attack_config)

    def run(self):
        """
        This method is called to perform the actual attack. You can access the self.
↪attack_config
```

```
        that holds the dict/json used when calling the endpoint to plan the attacks.
        """
        pass

    @staticmethod
    def to_dict():
        return Attack._to_dict(MyAttack.ref, MyAttack.schema, MyAttack.example)
```

Adding Custom Drivers

In order to interact with the cloud provider, you can use apache-libcloud, which is included as a dependency, to get some level of abstraction and reusability. The Chaos Monkey Engine provides with one driver out-of-the-box, the `chaosmonkey.drivers.EC2DriverFactory()`, that can be reused in your attacks or serve as inspiration:

```
class EC2DriverFactory:
    """
    Driver factory to get a libcloud driver with appropriate credentials for AWS␣
↪provider
    You can provide credentials by either:

    * Setting AWS_IAM_ROLE in env variables
    * Setting AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY in env variables

    You can provide the region to connect by either:

    * Provide it at instantiation time
    * Setting AWS_DEFAULT_REGION in env variables
    """

    def __init__(self, region=None):
        """
        Initialize an EC2 driver factory for a certain AWS region

        :param region: The AWS region to operate within
        :type region: string
        """

        self.IAM_METADATA_URL = "http://169.254.169.254/latest/meta-data/iam/security-
↪credentials"

        # First check if AWS_IAM_ROLE is defined
        aws_iam_role = os.environ.get("AWS_IAM_ROLE", None)
        if aws_iam_role is not None:
            # Get credentials from IAM role
            self.aws_ak, self.aws_sk, self.token = self._get_aws_credentials_from_iam_
↪role(aws_iam_role)
        else:
            # Get credentials from environment variables
            self.aws_ak = os.environ.get('AWS_ACCESS_KEY_ID')
            self.aws_sk = os.environ.get('AWS_SECRET_ACCESS_KEY')
            self.region = region if region is not None else os.environ.get("AWS_
↪DEFAULT_REGION")

    def get_driver(self):
        """
```

```
        Return a Libcloud driver for AWS EC2 Provider

        :return: Compute driver
        :type driver: Libcloud compute driver
        """
        return (get_driver(Provider.EC2))(self.aws_ak, self.aws_sk, region=self.
↪region)


    def _get_aws_credentials_from_iam_role(self, role):
        """
        With a valid IAM_ROLE make a request to the AWS metadata server to
        get temporary credentials for the role

        :param role: The IAM role to use
        :type role: string
        """
        url = "/".join((self.IAM_METADATA_URL, role))
        log.info("get aws credentials from AWS_IAM_ROLE (%s)", url)
        response = requests.get(url)
        response.raise_for_status()
        resp_json = response.json()
        aws_ak = resp_json.get("AccessKeyId")
        aws_sk = resp_json.get("SecretAccessKey")
        aws_token = resp_json.get("Token")
        return aws_ak, aws_sk, aws_token
```

CHAPTER 6

Testing

There are two type of tests, unit and acceptance. Both are put in the root /test folder

- **Unit Test** are build using PyTest
- **Acceptance Test** are build using Behave

There is a lot of work to do with testing and the coverage is not very high, however the infrastructure to create test is ready and some examples and guides can be found in the /test folder in the project.

We're using Tox to automate tests:

```
// run pylint, flake8, pytest and behave test
> tox
// generate documentation
> tox -e docs
// run bandit audit
> tox -e bandit
```

CHAPTER 7

API

Versions

The current API version is 1. You can add more versions and endpoints through the module `chaosmonkey.api()`

Authorization

The API is not protected with auth so every endpoint is publicly accessible

Date formats and timezone

Dates are always in the same format YYYY-MM-DDTHH:mm:ss. Valid dates are

- 2017-01-25T10:12:148
- 2016-11-05T18:12:148

When running the CME one of the configuration options is the **timezone**. Refer to *Usage*

Endpoints

Attacks Endpoints

Base path: /api/1/attacks

Attacks are python modules (located in /attacks folder) that are executed to perform actual attacks.

Each attack has three main properties represented in the API:

1. **example**: a JSON example for the attack. Use it as a template to call /plans endpoints

2. **ref**: its unique identifier. module_name:AttackClass
3. **schema**: json schema that validates the json representation for the attack

GET /api/1/attacks/
Return a list with the available attacks and its configuration.

Example:


```
{
    "attacks": [
        {
            "example": {
                "args": {
                    "filters": {
                        "tag:Name": "playground-asg"
                    },
                    "region": "eu-west-1"
                },
                "ref": "terminate_ec2_instance:TerminateEC2Instance"
            },
            "ref": "terminate_ec2_instance:TerminateEC2Instance",
            "schema": {
                "type": "object",
                "properties": {
                    "args": {
                        "type": "object",
                        "properties": {
                            "filters": {
                                "type": "object",
                                "properties": {
                                    "tag:Name": {
                                        "type": "string"
                                    }
                                },
                                "required": [
                                    "tag:Name"
                                ]
                            },
                            "region": {
                                "optional": true,
                                "type": "string"
                            }
                        },
                        "required": [
                            "region",
                            "filters"
                        ]
                    },
                    "ref": {
                        "type": "string"
                    }
                }
            }
        }
    ],
    "_links": {
        "self": {
            "href": "/api/1/attacks/"
        }
```

```
    }
}
```

Return `chaosmonkey.api.hal.document()`

Planners Endpoints

Base path: /api/1/planners

Planners are python modules (located in /planners folder). Planners are responsible of create *executors*.

Planners has three main properties represented in the API:

1. **example**: a JSON example for the planner
2. **ref**: its unique identifier. module_name:PlannerClass
3. **schema**: json schema that validates the planner

GET /api/1/planners/
Return a list with the available planners and its configuration.

Example response:

```
{
    "_links": {
        "self": {
            "href": "/api/1/planners/"
        }
    },
    "planners": [
        {
            "example": {
                "args": {
                    "times": 4,
                    "max_time": "15:00",
                    "min_time": "10:00"
                },
                "ref": "simple_planner:SimplePlanner"
            },
            "ref": "simple_planner:SimplePlanner",
            "schema": {
                "type": "object",
                "properties": {
                    "args": {
                        "type": "object",
                        "properties": {
                            "times": {
                                "type": "number"
                            },
                            "max_time": {
                                "type": "string"
                            },
                            "min_time": {
                                "type": "string"
                            }
                        }
                    },
                    "ref": {
```

```
                        "type": "string"
                    }
                }
            }
        }
    ]
}
```

Return `chaosmonkey.api.hal.document()`

Plans Endpoints

Base path: /api/1/plans

Plans receive a planner and an attack and create executors calling the corresponding planner with the given attack.

Each plan creates N executors related to an attack to be executed in the future.

Plans has the following properties

- **id**: unique identifier for a plan
- **created**: creation date
- **next_execution**: execution date for the next executor
- **name**: plan name
- **executors_count**: number of executors in the plan
- **executed**: if all the executors in the plan has been executed

`POST /api/1/plans/`

Add a plan.

Example request:

```
PUT /api/1/executors/3b373155577b4d1bbc62216ffea013a4
Body:
    {
        "name": "Terminate instances in Playground",
        "attack": {
            "args": {
                "region": "eu-west-1",
                "filters": {
                    "tag:Name": "playground-asg"
                }
            },
            "ref": "terminate_ec2_instance:TerminateEC2Instance"
        },
        "planner": {
            "ref": "simple_planner:SimplePlanner",
            "args": {
                "min_time" : "10:00",
                "max_time" : "19:00",
                "times": 4
            }
        }
    }
```

GET /api/1/plans/
List all plans created

Example request:

```
GET /api/1/plans/?all=true
```

Example response:

```
{
    "_links": {
        "self": {
            "href": "/api/1/plans/"
        }
    },
    "plans": [
        {
            "id": "6890192d8b6c40e5af16f13aa036c7dc",
            "created": "2017-01-26T10:41:1485427282",
            "next_execution": "2017-01-26 13:14:07.583372",
            "name": "Terminate instances in Playground",
            "executors_count": 2,
            "_links": {
                "self": {
                    "href": "/api/1/plans/6890192d8b6c40e5af16f13aa036c7dc"
                },
                "update": {
                    "href": "/api/1/plans/6890192d8b6c40e5af16f13aa036c7dc"
                },
                "delete": {
                    "href": "/api/1/plans/6890192d8b6c40e5af16f13aa036c7dc"
                }
            }
        }
    ]
}
```

Param all. Control when to show all plans (true) or only not executed (false). Defaults to false

Return `chaosmonkey.api.hal.document()`

GET /api/1/plans/ (**string:** *plan_id*)
Get a plan with all related executors

Example request:

```
GET /api/1/plans/6890192d8b6c40e5af16f13aa036c7dc
```

Example response:

```
{
    "id": "6890192d8b6c40e5af16f13aa036c7dc",
    "_embedded": {
        "executors": [
            {
                "plan_id": "6890192d8b6c40e5af16f13aa036c7dc",
                "_links": {
                    "self": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dcdd2530572fd04c5aa061f261f82743d3"
```

```
                    },
                    "update": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dcdd2530572fd04c5aa061f261f82743d3"
                    },
                    "delete": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dcdd2530572fd04c5aa061f261f82743d3"
                    }
                },
                "next_run_time": "2017-01-26T13:14:1485436447",
                "id": "dd2530572fd04c5aa061f261f82743d3"
            },
            {
                "plan_id": "6890192d8b6c40e5af16f13aa036c7dc",
                "_links": {
                    "self": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dc1dd3f0d392e545808edb74852213c1ae"
                    },
                    "update": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dc1dd3f0d392e545808edb74852213c1ae"
                    },
                    "delete": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dc1dd3f0d392e545808edb74852213c1ae"
                    }
                },
                "next_run_time": "2017-01-26T18:24:1485455082",
                "id": "1dd3f0d392e545808edb74852213c1ae"
            }
        ]
    },
    "created": "2017-01-26T10:41:1485427282",
    "next_execution": null,
    "name": "Terminate instances in Playground",
    "executors_count": null,
    "_links": {
        "self": {
            "href": "/api/1/plans/6890192d8b6c40e5af16f13aa036c7dc"
        }
    }
}
```


Return `chaosmonkey.api.hal.document()`

`DELETE /api/1/plans/` (**string:** *plan_id*)

Delete a plan

Example request:

```
DEL /api/1/plans/6890192d8b6c40e5af16f13aa036c7dc
```

Executors Endpoints

Base path: /api/1/executors

Executors are scheduled jobs that are related with an attack, so in the given date the job will execute the attack. The only way to create executors is through *chaosmonkey.api.plans_blueprint()*

Evey executor has 4 main properties:

1. **id**: unique identifier
2. **next_run_time**: The time and date that the executor is going to be executed
3. **plan_id**: id of the plan that created the executor
4. **executed**: if the executor has been executed

Example:

```
{
    "id": "3b373155577b4d1bbc62216ffea013a4",
    "plan_id": "3ec72048cab04b76bdf2cfd4bc81cd1e",
    "next_run_time": "2017-01-25T10:12:1485339145",
    "executed": false
}
```

GET /api/1/executors/
Get a list of scheduled executors

Example response:

```
{
    "executors":[
        {
            "_links":{
                "self":{
                    "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
                },
                "update":{
                    "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
                },
                "delete":{
                    "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
                }
            },
            "id":"3b373155577b4d1bbc62216ffea013a4",
            "plan_id":"3ec72048cab04b76bdf2cfd4bc81cd1e",
            "next_run_time":"2017-01-25T10:12:1485339145",
            "executed": false
        }
    ]
}
```

Param executed. Control when to show all executors (true) or only not executed (false). Defaults to false

Return `chaosmonkey.api.hal.document()`

PUT /api/1/executors/ (**string:** *executor_id*)
Update a executor to change its date. To provide a new date use the format in the example bellow. The format is used to create a DateTrigger from the apscheduler.

TODO: create more Triggers

Example request:

```
PUT /api/1/executors/3b373155577b4d1bbc62216ffea013a4
Body:
    {
      "type" : "date",
      "args" : {
        "date": "2017-10-23T19:19"
      }
    }
```

Example response:

```
{
  "id": "3b373155577b4d1bbc62216ffea013a4",
  "plan_id": "3ec72048cab04b76bdf2cfd4bc81cd1e",
  "next_run_time": "2017-10-23T19:19:1508786354",
  "executed": false,
  "_links": {
    "self": {
      "href": "/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
    },
    "update":{
      "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
    },
    "delete":{
      "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
    }
  }
}
```

Return `chaosmonkey.api.hal.document()`

DELETE /api/1/executors/ (**string:** *executor_id*)
Delete an executor

Example request:

```
DEL /api/1/executors/6890192d8b6c40e5af16f13aa036c7dc
```

CHAPTER 8

chaosmonkey

chaosmonkey package

Subpackages

chaosmonkey.api package

Submodules

chaosmonkey.api.api_errors module

This module contains an object that represent an API Error

Any APIError thrown in an endpoint is handled to return to the user a proper json error with custom status code and message

exception `chaosmonkey.api.api_errors.`**`APIError`**(*message*, *status_code=None*, *payload=None*)

Bases: `Exception`

Represents an API Error

Parameters

- **`message`** – message to be returned to the user
- **`status_code`** – response status code (defaults to 400)
- **`payload`** – custom payload to give extra info in the response

Example:

```
>>> raise APIError("Error detected", 500, {"extra": "extra_info"})
```

to_dict()
Convert exception to dict

chaosmonkey.api.request_validator module

chaosmonkey.api.request_validator.**validate_payload**(*request*, *schema*)
validates a request payload against a json schema

Parameters

- **request** – request received with valid json body
- **schema** – schema to validate the request payload

Returns True

Raises *chaosmonkey.api.api_errors()*

chaosmonkey.api.attacks_blueprint module

Base path: /api/1/attacks

Attacks are python modules (located in /attacks folder) that are executed to perform actual attacks.

Each attack has three main properties represented in the API:

1. **example**: a JSON example for the attack. Use it as a template to call /plans endpoints
2. **ref**: its unique identifier. module_name:AttackClass
3. **schema**: json schema that validates the json representation for the attack

chaosmonkey.api.attacks_blueprint.**list_attacks**()
Return a list with the available attacks and its configuration.

Example:


```
{
    "attacks": [
        {
            "example": {
                "args": {
                    "filters": {
                        "tag:Name": "playground-asg"
                    },
                    "region": "eu-west-1"
                },
                "ref": "terminate_ec2_instance:TerminateEC2Instance"
            },
            "ref": "terminate_ec2_instance:TerminateEC2Instance",
            "schema": {
                "type": "object",
                "properties": {
                    "args": {
                        "type": "object",
                        "properties": {
                            "filters": {
                                "type": "object",
                                "properties": {
                                    "tag:Name": {
```

```
                                        "type": "string"
                                    }
                                },
                                "required": [
                                    "tag:Name"
                                ]
                            },
                            "region": {
                                "optional": true,
                                "type": "string"
                            }
                        },
                        "required": [
                            "region",
                            "filters"
                        ]
                    },
                    "ref": {
                        "type": "string"
                    }
                }
            }
        }
    ],
    "_links": {
        "self": {
            "href": "/api/1/attacks/"
        }
    }
}
```

Returns `chaosmonkey.api.hal.document()`

chaosmonkey.api.executors_blueprint module

Base path: /api/1/executors

Executors are scheduled jobs that are related with an attack, so in the given date the job will execute the attack. The only way to create executors is through *chaosmonkey.api.plans_blueprint()*

Evey executor has 4 main properties:

1. **id**: unique identifier
2. **next_run_time**: The time and date that the executor is going to be executed
3. **plan_id**: id of the plan that created the executor
4. **executed**: if the executor has been executed

Example:

```
{
    "id": "3b373155577b4d1bbc62216ffea013a4",
    "plan_id": "3ec72048cab04b76bdf2cfd4bc81cd1e",
    "next_run_time": "2017-01-25T10:12:1485339145",
    "executed": false
}
```

chaosmonkey.api.executors_blueprint.**delete_executor**(*executor_id*)
Delete an executor

Example request:

```
DEL /api/1/executors/6890192d8b6c40e5af16f13aa036c7dc
```

chaosmonkey.api.executors_blueprint.**dict_to_trigger**(*trigger_dict*)
Returns a trigger version of the trigger json

chaosmonkey.api.executors_blueprint.**get_executors**()
Get a list of scheduled executors

Example response:

```
{
    "executors":[
        {
            "_links":{
                "self":{
                    "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
                },
                "update":{
                    "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
                },
                "delete":{
                    "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
                }
            },
            "id":"3b373155577b4d1bbc62216ffea013a4",
            "plan_id":"3ec72048cab04b76bdf2cfd4bc81cd1e",
            "next_run_time":"2017-01-25T10:12:1485339145",
            "executed": false
        }
    ]
}
```

Param executed. Control when to show all executors (true) or only not executed (false). Defaults to false

Returns chaosmonkey.api.hal.document()

chaosmonkey.api.executors_blueprint.**put_executor**(*executor_id*)
Update a executor to change its date. To provide a new date use the format in the example bellow. The format is used to create a DateTrigger from the apscheduler.

TODO: create more Triggers

Example request:

```
PUT /api/1/executors/3b373155577b4d1bbc62216ffea013a4
Body:
    {
      "type" : "date",
      "args" : {
        "date": "2017-10-23T19:19"
      }
    }
```

Example response:

```
{
  "id": "3b373155577b4d1bbc62216ffea013a4",
  "plan_id": "3ec72048cab04b76bdf2cfd4bc81cd1e",
  "next_run_time": "2017-10-23T19:19:1508786354",
  "executed": false,
  "_links": {
    "self": {
      "href": "/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
    },
    "update":{
      "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
    },
    "delete":{
      "href":"/api/1/executors/3b373155577b4d1bbc62216ffea013a4"
    }
  }
}
```

Returns `chaosmonkey.api.hal.document()`

`chaosmonkey.api.executors_blueprint.`**`trigger_to_dict`**(*trigger*)
Returns a dict version of the trigger

chaosmonkey.api.planners_blueprint module

Base path: /api/1/planners

Planners are python modules (located in /planners folder). Planners are responsible of create *executors*.

Planners has three main properties represented in the API:

1. **example**: a JSON example for the planner
2. **ref**: its unique identifier. module_name:PlannerClass
3. **schema**: json schema that validates the planner

`chaosmonkey.api.planners_blueprint.`**`list_planners`**()
Return a list with the available planners and its configuration.

Example response:

```
{
    "_links": {
        "self": {
            "href": "/api/1/planners/"
        }
    },
    "planners": [
        {
            "example": {
                "args": {
                    "times": 4,
                    "max_time": "15:00",
                    "min_time": "10:00"
                },
                "ref": "simple_planner:SimplePlanner"
```

```
            },
            "ref": "simple_planner:SimplePlanner",
            "schema": {
                "type": "object",
                "properties": {
                    "args": {
                        "type": "object",
                        "properties": {
                            "times": {
                                "type": "number"
                            },
                            "max_time": {
                                "type": "string"
                            },
                            "min_time": {
                                "type": "string"
                            }
                        }
                    },
                    "ref": {
                        "type": "string"
                    }
                }
            }
        }
    ]
}
```

Returns `chaosmonkey.api.hal.document()`

chaosmonkey.api.plans_blueprint module

Base path: /api/1/plans

Plans receive a planner and an attack and create executors calling the corresponding planner with the given attack.

Each plan creates N executors related to an attack to be executed in the future.

Plans has the following properties

- **id**: unique identifier for a plan
- **created**: creation date
- **next_execution**: execution date for the next executor
- **name**: plan name
- **executors_count**: number of executors in the plan
- **executed**: if all the executors in the plan has been executed

`chaosmonkey.api.plans_blueprint.`**`add_plan`**`()`

Add a plan.

Example request:

```
PUT /api/1/executors/3b373155577b4d1bbc62216ffea013a4
Body:
    {
```

```
        "name": "Terminate instances in Playground",
        "attack": {
            "args": {
                "region": "eu-west-1",
                "filters": {
                    "tag:Name": "playground-asg"
                }
            },
            "ref": "terminate_ec2_instance:TerminateEC2Instance"
        },
        "planner": {
            "ref": "simple_planner:SimplePlanner",
            "args": {
                "min_time" : "10:00",
                "max_time" : "19:00",
                "times": 4
            }
        }
    }
```

chaosmonkey.api.plans_blueprint.**delete_plan**(*plan_id*)

Delete a plan

Example request:

```
DEL /api/1/plans/6890192d8b6c40e5af16f13aa036c7dc
```

chaosmonkey.api.plans_blueprint.**get_plan**(*plan_id*)

Get a plan with all related executors

Example request:

```
GET /api/1/plans/6890192d8b6c40e5af16f13aa036c7dc
```

Example response:

```
{
    "id": "6890192d8b6c40e5af16f13aa036c7dc",
    "_embedded": {
        "executors": [
            {
                "plan_id": "6890192d8b6c40e5af16f13aa036c7dc",
                "_links": {
                    "self": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dcdd2530572fd04c5aa061f261f82743d3"
                    },
                    "update": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dcdd2530572fd04c5aa061f261f82743d3"
                    },
                    "delete": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dcdd2530572fd04c5aa061f261f82743d3"
                    }
                },
                "next_run_time": "2017-01-26T13:14:1485436447",
                "id": "dd2530572fd04c5aa061f261f82743d3"
```

```
            },
            {
                "plan_id": "6890192d8b6c40e5af16f13aa036c7dc",
                "_links": {
                    "self": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dc1dd3f0d392e545808edb74852213c1ae"
                    },
                    "update": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dc1dd3f0d392e545808edb74852213c1ae"
                    },
                    "delete": {
                        "href": "/api/1/plans/
↪6890192d8b6c40e5af16f13aa036c7dc1dd3f0d392e545808edb74852213c1ae"
                    }
                },
                "next_run_time": "2017-01-26T18:24:1485455082",
                "id": "1dd3f0d392e545808edb74852213c1ae"
            }
        ]
    },
    "created": "2017-01-26T10:41:1485427282",
    "next_execution": null,
    "name": "Terminate instances in Playground",
    "executors_count": null,
    "_links": {
        "self": {
            "href": "/api/1/plans/6890192d8b6c40e5af16f13aa036c7dc"
        }
    }
}
```

Returns chaosmonkey.api.hal.document()

chaosmonkey.api.plans_blueprint.**list_plans**()

List all plans created

Example request:

```
GET /api/1/plans/?all=true
```

Example response:

```
{
    "_links": {
        "self": {
            "href": "/api/1/plans/"
        }
    },
    "plans": [
        {
            "id": "6890192d8b6c40e5af16f13aa036c7dc",
            "created": "2017-01-26T10:41:1485427282",
            "next_execution": "2017-01-26 13:14:07.583372",
            "name": "Terminate instances in Playground",
            "executors_count": 2,
```

```
            "_links": {
                "self": {
                    "href": "/api/1/plans/6890192d8b6c40e5af16f13aa036c7dc"
                },
                "update": {
                    "href": "/api/1/plans/6890192d8b6c40e5af16f13aa036c7dc"
                },
                "delete": {
                    "href": "/api/1/plans/6890192d8b6c40e5af16f13aa036c7dc"
                }
            }
        }
    ]
}
```

Param all. Control when to show all plans (true) or only not executed (false). Defaults to false

Returns `chaosmonkey.api.hal.document()`

Module contents

API Package.

This package creates a flask application with the API to interact with the CME.

chaosmonkey.attacks package

Submodules

chaosmonkey.attacks.attack module

class `chaosmonkey.attacks.attack.`**`Attack`**(*attack_config*)

Bases: `object`

Base class for attacks. Every attack must extend from this class

`example` = None
dict example for using when calling add plan endpoint

`ref` = None
string Unique identifier for the attack. Must be module_name.AttackClass

`run`()
This method is called by an executor to perform the actual attack. You can access the self.attack_config that holds the configuration used when calling the endpoint to plan the attacks.

`schema` = None
dict Valid jsonSchema to validate the attack attributes in the API

static `to_dict`()
You should implement to_dict to return an Attack._to_dict(ref, schema, example) using the attack attributes

Example:

```
@staticmethod
def to_dict():
    return Attack._to_dict(
        TerminateEC2Instance.ref,
        TerminateEC2Instance.schema,
        TerminateEC2Instance.example
    )
```

chaosmonkey.attacks.executor module

chaosmonkey.attacks.executor.**execute**(*attack_config=None*, *plan_id=None*)
This func is executed for every job stored in the scheduler. Receive in kwargs all attack configuration used when creating the executor that indicates which attack and configuration should be used to do the actual attack.

Parameters

- **attack_config** – **Dict** with attack configuration
- **plan_id** – **String** plan id for the plan containing the executor

Module contents

chaosmonkey.cm package

Submodules

chaosmonkey.cm.cm module

chaosmonkey.cm.cm.**run_api_server**(*app*, *port=5000*)
Runs a gevent server on a given port. Default port is 5000.

Parameters

- **app** – WSGI-compliant app to be run
- **port** (*int*) – port for the gevent server

Returns None

chaosmonkey.cm.cm.**sigterm_handler**()

Module contents

chaosmonkey.dal package

Submodules

chaosmonkey.dal.cme_sqlalchemy_store module

CMESQLAlchemyStore replaces the default SQLAlchemyJobstore from apscheduler

It controls the persistence layer.

class chaosmonkey.dal.cme_sqlalchemy_store.**CMESQLAlchemyStore**(*pickle_protocol=4*)
Bases: apscheduler.jobstores.base.BaseJobStore

Manage persistence for apscheduler and CMEEngine.

This class is used by the apscheduler, all overridden methods should return appscheduler.job.Job objects.

The store handles 2 types of models: plans and executors. Internally apscheduler names the executors as jobs.

TODO: executors are marked as executed when they are processed (no mather if they fail or succeed. We need to handle execution errors in order to know whats going on with the execution and what was its result.)

- Plans: *chaosmonkey.dal.plan_model.Plan()*
- Executors: *chaosmonkey.dal.executor_model.Executor()*

add_job(*job*)

add_plan(*name*)
Create a plan in the db.

Parameters **name** – string

Returns Plan created

delete_plan(*plan_id*)
Delete a plan.

All the executors related to the plan are deleted. (ON_DELETE constrain in db.Models)

Parameters **plan_id** – string

get_all_jobs()

get_due_jobs(*now*)

get_executor(*executor_id*)
Get an executor

Parameters **executor_id** – string

Returns List of Executor

get_executors(*executed=False*)
Get a list of executors

Returns List of Executor

get_executors_for_plan(*plan_id*)
Get a list of executors related to a plan by its plan_id

Parameters **plan_id** – string

Returns List of Executor

get_next_run_time()

get_plan(*plan_id*)
Return a plan by its id

Parameters **plan_id** – string

Returns Plan

get_plans(*show_all=False*)
Return a list of plans created on db. For each plan return the number of pending executors and the next_run_time of the first executor

Returns List of Plans

lookup_job(*job_id*)

real_remove_job(*job_id*)

remove_all_jobs()

remove_job(*job_id*)
Instead of deleting a job when its executed or it has failed, check it as executed.

TODO: delete the executor and save to a historic table the executed attacks with its logs and results.

shutdown()

start(*scheduler*, *alias*)
Start the SQLAlchemy engine

update_job(*job*)

exception chaosmonkey.dal.cme_sqlalchemy_store.**PlanLookupError**(*job_id*)
Bases: KeyError

Raised when the store cannot find a plan for update or removal.

chaosmonkey.dal.database module

SQLAlchemy database

chaosmonkey.dal.executor_model module

class chaosmonkey.dal.executor_model.**Executor**(*job_id*, *next_run_time*, *plan_id*, *job_state=None*)
Bases: flask_sqlalchemy.Model

Executors are persistent representations of scheduled jobs. This model is shared between the cme and apscheduler.

executed
if the job was executed

id
unique identifier

job_state
store the full state of the executor (with pickle)

next_run_time
DateTime for the executor to be executed

plan_id
plan id reference

to_dict()
Return a chaosmonkey.api.hal.document() representation for the Executor

Returns *chaosmonkey.dal.executor_model.HalExecutor()*

class chaosmonkey.dal.executor_model.**HalExecutor**(*data=None*, *links=None*, *embedded=None*)
Bases: chaosmonkey.api.hal.BaseDocument

Class to represent an Executor as a chaosmonkey.api.hal.document()

chaosmonkey.dal.plan_model module

class chaosmonkey.dal.plan_model.**HalPlan**(*data=None*, *links=None*, *embedded=None*)
Bases: chaosmonkey.api.hal.BaseDocument

Class to represent a Plan as a chaosmonkey.api.hal.document()

class chaosmonkey.dal.plan_model.**Plan**(*_id=None*, *name=None*, *created=None*, *next_execution=None*, *executors_count=0*, *executed=False*)
Bases: flask_sqlalchemy.Model

Store information about the plan and its executors.

This model is only used by the cme.

created
creation datetime

executed
if all the executors in the plan has been executed

executors_count = None
number of pending executors

id
unique identifier

jobs

name
plan name

next_execution = None
DateTime for the next executor execution time

to_dict()
Returns a chaosmonkey.api.hal.document() representation for the Executor

Returns *chaosmonkey.dal.plan_model.HalPlan()*

Module contents

Data Access Layer package.

Modules related to data access

chaosmonkey.engine package

Submodules

chaosmonkey.engine.app module

ChaosMonkey Engine

chaosmonkey.engine.app.**configure_engine**(*database_uri*, *attacks_folder*, *planners_folder*, *cme_timezone*)

Create a Flask App and all the configuration needed to run the CMEEngine

- Init and configure the SQLAlchemy store (create db and tables if don't exists)
- Init ModuleStores (attacks and planners)
- Configure the timezone and jobstore for the scheduler
- Configure the CMEManager

TODO: The scheduler start is not made until the first request is made. This is due to the way the SQLAlchemy store is created, because it needs the app.context to work properly

Parameters

- **database_uri** – SQLAlchemy SQLALCHEMY_DATABASE_URI
- **attacks_folder** – folder to load the attacks modules
- **planners_folder** – folder to load the planners modules
- **cme_timezone** – timezone to set in the scheduler

chaosmonkey.engine.app.**make_sure_path_exists**(*path*)

Make sure a path exists and create it if don't

Parameters **path** – string path to check

chaosmonkey.engine.app.**shutdown_engine**()

Shutdown the scheduler

chaosmonkey.engine.app.**start_scheduler**()

chaosmonkey.engine.cme_manager module

CME Manager

Control layer for CME Engine

class chaosmonkey.engine.cme_manager.**CMEManager**

Bases: object

CMEManager is the manager responsible of communicating the API with the backend.

It manages:

- scheduler: BackgroundScheduler from appscheduler lib. The scheduler run executors
- sql_store: SQLAlchemy store. The persistence layer
- planners_store: ModuleStore that load and manages available planners.
- attacks_store: ModuleStore that load and manage available attacks.

Methods to interact with Executors and Plans always returns the db.Models (chaosmonkey.dal.*_model)

Methods that interact with Attacks always returns Attacks objects (chaosmonkey.attacks.attack)

Methods that interact with Planners always returns Planners objects (chaosmonkey.planners.planner)

add_executor(*date*, *name*, *attack_config*, *plan_id*)
Adds a new executor to the scheduler

Parameters

- **date** – Datetime to execute the job
- **name** – Executor name
- **attack_config** – Attack config. Dict to be passed to the executor on execution time
- **plan_id** – Referenced plan id

Returns chaosmonkey.dal.executor.Executor

add_plan(*name*)
Creates a new plan in the sqlStore

Parameters **name** – Plan name

Returns chaosmonkey.dal.plan.Plan

attacks_store
Attacks store property

configure(*scheduler*, *sql_store*, *planners_store*, *attacks_store*)
Configure the manager

Parameters

- **scheduler** – apscheduler.schedulers.background.BackgroundScheduler
- **sql_store** – SQLAlchemy
- **planners_store** – chaosmonkey.modules.ModuleStore
- **attacks_store** – chaosmonkey.modules.ModuleStore

Returns

delete_plan(*plan_id*)
Delete a plan (and all his associated executors) from the sqlStore

Parameters **plan_id** – String plan Id

Returns

execute_plan(*name*, *planner_config*, *attack_config*)
Execute a plan with a planner and executor config to create executors based on the configs

It also validates the planner and executor config against the modules

Parameters

- **name** – Plan name
- **planner_config** – Dict with planner config
- **attack_config** – Dict with attack config

get_attack_list()
Return a list with all attacks loaded in the self._attacks_store

Returns chaosmonkey.attacks.attack.Attack list

get_executor(*executor_id*)
Return an Executor object with the given id :return: chaosmonkey.dal.executor.Executor

get_executors(*executed=False*)
Return a list of Executor objects created in DB :return: chaosmonkey.dal.executor.Executor list

get_executors_for_plan(*plan_id*)
Return a list of Executors for a given plan id :return: chaosmonkey.dal.executor.Executor

get_plan(*plan_id*)
Returns a plans

Returns chaosmonkey.dal.plan.Plan

get_planner_list()
Return a list with all planners loaded in the self._planners_store

Returns chaosmonkey.planners.planner.Planner list

get_plans(*show_all=None*)
Returns a list with al plans in the sqlStore

Returns List of chaosmonkey.dal.plan.Plan

planners_store
Planners store property

remove_executor(*executor_id*)
Removes an executor by his ID :param executor_id: :return:

scheduler
Scheduler property

sql_store
SQLstore property

update_executor_trigger(*executor_id*, *trigger*)
Update an executor trigger to change is due date

Parameters

- **executor_id** – executor id
- **trigger** – apscheduler.triggers.BaseTrigger instance

Returns chaosmonkey.dal.executor.Executor

chaosmonkey.engine.scheduler module

`apscheduler.schedulers.background.BackgroundScheduler()` used by the CME.

The scheduler is responsible of storing executors and execute them in the given datatime.

Module contents

This package contains the pieces to run the Engine

- Engine configuration (FlaskApp, Store, Modules...)
- CMEManager, used to communicate the Engine with the API and the Store
- Scheduler to schedule executors for later use

chaosmonkey.modules package

Submodules

chaosmonkey.modules.module_store module

ModulesStore Handle the dynamic load of modules outside the main package.

exception chaosmonkey.modules.module_store.**ModuleLookupError**(*message*)
Bases: Exception

Represent a Module lookup error when a module is not found by its ref

class chaosmonkey.modules.module_store.**ModulesStore**(*klass*)
Bases: object

Load and store modules from outside the app package

add(*module*)

get(*ref*)
Given a str reference of a class in a module (eg. ModuleName.ClassName) return the class so it can be instantiated

The module with the class must be loaded first using load or add method.

Parameters **ref** – str representation of a class in a module

Returns class ready for instantation

get_modules()

list()
List all loaded modules names. It will return a list withe the __name__ of each module :return:

load(*path*)
Loads all modules found in a given path. It adds the path to the sys.path and import all modules found

Parameters **path** – path for lookup modules

modules = []

remove(*module_name*)

set_modules(*modules*)

Module contents

chaosmonkey.planners package

Submodules

chaosmonkey.planners.planner module

Base class for planners

Every planner must extend Planner class

class chaosmonkey.planners.planner.**Planner**(*name*)
Bases: object

Planner interface Planners are responsible for scheduling jobs that executes attacks

Parameters **name** – plan name

static add_plan(*name*)

example = None

plan(*planner_config*, *attack_config*)
Plan the jobs. This method should use the config to schedule jobs based on the configuration for the planner

Parameters

- **planner_config** – configuration related to the scheduler
- **attack_config** – configuration related to the attack

ref = None

schema = None

static to_dict()

Module contents

chaosmonkey.drivers package

Submodules

chaosmonkey.drivers.planner module

class chaosmonkey.drivers.ec2_driver.**EC2DriverFactory**(*region=None*)
Bases: object

Driver factory to get a libcloud driver with appropriate credentials for AWS provider You can provide credentials by either:

•Setting AWS_IAM_ROLE in env variables

•Setting AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY in env variables

You can provide the region to connect by either:

•Provide it at instantiation time

•Setting AWS_DEFAULT_REGION in env variables

get_driver()
Return a Libcloud driver for AWS EC2 Provider

Returns Compute driver

Module contents

Module contents

Chaos Monkey Engine main package

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

HTTP Routing Table

/api

Index

E

G

H

I

J

L