deeptracy Documentation

Release 0.0.7

BBVA

Nov 15, 2017

Contents

Attention: This documentation is a work in progress. We are currently in development phase and changing things in a daily basis.

Deeptracy is a tool that can scan projects to find vulnerabilities in its dependencies. It works by accessing the source code of repositories and extracting the dependencies list to match them against the NIST NVD Data Feeds

Deeptracy perks:

- Deployed as docker containers
- Scalable
- Usable inside deployment pipelines
- Multi-language (Scan projects in Python, Java, Javascript and more)
- Reactive (we monitor new vulnerabilities and warn you if any affects your dependencies)
- Open Source :D

CHAPTER 1

Welcome to Deeptracy

Welcome to Deeptracy's documentation. This documentation is divided into two different parts. One is the userdocs-ref which include installation and usage, and the other is the *Developer's Documentation* which include sourcecode-ref documentation, local environment, testing and so on.

CHAPTER 2

User's Documentation

This documentation is for users who want to use Deeptracy. It covers two parts, *Installation* and *Usage*

2.1 Installation

2.1.1 Components

Deeptracy has four main components:

- *Deeptracy Workers* is responsible of extracting project dependencies, cloning repositories, notifications and more.
- *Deeptracy API* this is the main entrance for actions
- *Deeptracy Dashboard* this is the dashboard for visual information on the system
- *Patton* is responsible to match dependencies with vulnerabilities

Each of this components is shipped as a docker image. You can find them in the public deeptracy dockerhub https://hub.docker.com/search/?isAutomated=0&isOfficial=0&page=1&pullCount=0&q=deeptracy&starCount=0.

Beside the components of Deeptracy, the system needs two more things to work:

- Postgres database to store projects, scans and so on
- Redis in-memory data structure store used as message broker

This two components can be launched as a docker containers, but you can also install them without docker.

Note: Note that if you want to run the containers by hand (no docker-compose) you need to create a custom network yourself. You can use docker-compose to run all deeptracy components at once (see *Bringing up the environment*)

```
$ docker network create deeptracy
$ docker run --network=deeptracy -p 5432:5432 -d --name=postgres -e POSTGRES_
↪PASSWORD=postgres postgres:alpine
$ docker run --network=deeptracy -p 6379:6379 -d --name=redis redis:3-alpine
```

2.1.2 Deeptracy Workers

Workers are celery processes. You can launch any number of workers **on the same hosts**. As they are celery workers connected to a broker (redis), they will take tasks to even the workload.

One of the tasks performed by the workers is cloning repositories. For this, you need to **mount the same volume in each worker from the host**, where the repositories will be cloned. This volume (*SHARED_VOLUME_PATH*) will be mounted in various containers that the worker uses to perform distinct tasks.

```
$ docker run -d -e BROKER_URI=redis://redis:6379 \
               -e DATABASE_URI=postgresql://postgres:postgres@postgres:5432/deeptracy␣
↪\
               -e SHARED_VOLUME_PATH=/tmp/deeptracy \
               -e PLUGINS_LOCATION=/opt/deeptracy/plugins \
               -v /tmp:/tmp \
               --network=deeptracy \
               bbvalabs/deeptracy:latest
```

Warning: Because the repository to scan is only downloaded once, you can't have workers on different hosts, as the source code for the project is only present int he hosts that perform the task to download it.

The workers performs almost all the task inside docker containers. The worker image has docker installed, but you can mount the docker socket from the host in to the worker containers, so the docker in the host would be used.

Environment Variables

This are the environment variables needed by the workers

- **BROKER_URI** Url to the redis broker (Ex. redis://127.0.0.1:6379)
- **DATABASE_URI** Url to the postgres database (Ex. postgresql://postgres:postgres@127.0.0.1:5432/deeptracy)
- **SHARED_VOLUME_PATH Path in the host to mount as a volume in Docker images. this folder** is going to be used to clone projects to be scanned. (Ex. /tmp/deeptracy)
- **LOCAL_PRIVATE_KEY_FILE If you wanna clone private repositories, you can specify a private key file to** be used when cloning such repos.

2.1.3 Deeptracy API

The API component provides the access point to interact with deeptracy.

```
docker run -d -e BROKER_URI=redis://redis:6379 \
               -e DATABASE_URI=postgresql://postgres:postgres@postgres:5432/deeptracy␣
↪\
               -e SERVER_ADDRESS=0.0.0.0:8080 \
               -e GUNICORN_WORKERS=5 \
               -p 8080:8080 \
```

```
                --network=deeptracy \
                bbvalabs/deeptracy-api:latest
```

2.1.4 Deeptracy Dashboard

With the dashboard you have a visual representation of the system. You can also access scan results, vulnerabilities and more.

```
docker run -d -e BROKER_URI=redis://redis:6379 \
                -e DATABASE_URI=postgresql://postgres:postgres@postgres:5432/deeptracy␣
↪\
                -e SERVER_ADDRESS=localhost:8080
                -p 8000:8000 \
                --network=deeptracy \
                bbvalabs/deeptracy-dashboard:latest
```

2.1.5 Patton

Path Patton is the responsible of matching dependencies with vulnerabilities. It has its own database (you can use a namespace in a shared postgresql database) and it has an auto-sync mechanism with the vulnerabilities database.

```
docker run -d -e BROKER_URI=redis://redis:6379 \
                -e DATABASE_URI=postgresql://postgres:postgres@postgres:5432/patton \
                -p 8001:8001 \
                --network=deeptracy \
                bbvalabs/patton:latest
```

2.1.6 Bringing up the environment

As all the pieces are shipped as Docker containers, is easy to bring up an environment. You can find an example with code to launch Deeptracy in a single AWS instance in the deploy folder.

This is an example of a complete Docker Compose file that launch a complete working environment.

```
version: '3'

services:
  deeptracy:
    image: bbvalabs/deeptracy
    depends_on:
      - redis
      - postgres
    environment:
      - BROKER_URI=redis://redis:6379
      - DATABASE_URI=postgresql://postgres:postgres@postgres:5432/deeptracy
      - SHARED_VOLUME_PATH=/tmp/deeptracy
      - LOCAL_PRIVATE_KEY_FILE=/tmp/id_rsa
      - PLUGINS_LOCATION=/opt/deeptracy/plugins
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /tmp:/tmp
    privileged: true
    command: ["./wait-for-it.sh", "postgres:5432", "--", "/opt/deeptracy/run.sh"]
```

```
deeptracy-api:
  image: bbvalabs/deeptracy-api
  depends_on:
    - redis
    - postgres
  ports:
    - 8080:8080
  environment:
    - BROKER_URI=redis://redis:6379
    - DATABASE_URI=postgresql://postgres:postgres@postgres:5432/deeptracy
    - SERVER_ADDRESS=0.0.0.0:8080
    - GUNICORN_WORKERS=1
    - LOG_LEVEL=DEBUG
  command: ["./wait-for-it.sh", "postgres:5432", "--", "/opt/deeptracy/run.sh"]

patton:
  image: bbvalabs/patton
  depends_on:
    - redis
    - postgres
  ports:
    - 8001:8001
  environment:
    - BROKER_URI=redis://redis:6379
    - DATABASE_URI=postgresql://postgres:postgres@postgres:5432/patton
    - LOG_LEVEL=DEBUG
  command: ["./wait-for-it.sh", "postgres:5432", "--", "/opt/deeptracy/run.sh"]

deeptracy-dashboard:
  image: bbvalabs/deeptracy-dashboard
  ports:
    - 80:8080
  environment:
    - SERVER_ADDRESS=localhost:8080

postgres:
  image: postgres:9.6-alpine
  ports:
    - 5432:5432
  environment:
    - POSTGRES_PASSWORD=postgres
  command: -p 5432

redis:
  image: redis:3-alpine
  ports:
    - 6379:6379
```

This docker compose will bring up an environment ready to be used. You can access the dashboard at `localhost`

2.2 Usage

This section explain how to use Deeptracy. Once installed Deeptracy can be used as a service. This means that a public API is exposed and all functionalities can be used through it.

2.2.1 Create Projects

Projects are the main object in the API. A project represents a single repository that you want to scan and monitorice for vulnerabilities. You can't have more that one project with the same repository in the database.

To create projects you need to invoke the *Create Projects* endpoint after any scan.

2.2.2 Launch Scans

Every time a scan is launched, Deeptracy will check for the project dependencies. **If the dependencies have changed from the last scan performed, the scan will begin**.

A scan is performed by cloning the project repository and running different plugins against the source code. You can launch scans manually by calling the *Create Scan* endpoint or by *Configuring a hook for your project*.

2.2.3 Spot Vulnerabilities

Evey scan will run N analyzers (one for each plugin available in the system) and save the vulnerabilities found on the database. Once all analyzers are done, all vulnerabilities are merged together and saved as a final vulnerability list.

You can access individual analyzer results with *Get Analyzer Vulnerabilities* endpoint or the final scan list with the *Get Scan Vulnerabilities* endpoint.

2.2.4 Get Notified

Every time a scan finishes, if your project have the information to receive notifications you will receive one with the spotted vulnerabilities.

2.2.5 Configuring a hook for your project

You can configure a hook in your repository, so every time a push is detected a scan will automatically launched for your project. The url for the hook is `{host}/api/1/webhook/`

- Github Create a webhook for *PUSH* actions only
- Bitbucket Create a webhook for *PUSH* actions only

2.3 API Reference

This section of the documentation exposes the API methods availables to interact with.

2.3.1 Create Projects

2.3.2 Create Scan

2.3.3 Get Analyzer Vulnerabilities

2.3.4 Get Scan Vulnerabilities

CHAPTER 3

Developer's Documentation

This documentation is for developers who want to contribute to Deeptracy.

3.1 Installation

3.1.1 Python Version

We recommend using the latest version of Python 3. Deeptracy supports Python 3.6 and newer.

3.1.2 Deeptracy Projects

Deeptracy has four repositories with each of its components:

- Workers main repository with celery tasks and plugins
- Api holds the Flask API
- Dashboard has the front web
- Core shared library between workers and api projects. Data access components and plugins perks.

For develop, is recommended that you clone each repository under the same work dir:

```
- deeptracy-project
|- deeptracy
|- deeptracy-api
|- deeptracy-core
|- deeptracy-dashboard
```

3.1.3 Virtual environments

Is highly recommended to work with a single virtual environment for all the projects by creating a single environment at the same level that the rest of the projects

```
- deeptracy-project
|- deeptracy
|- deeptracy-api
|- deeptracy-core
|- deeptracy-dashboard
|- .venv
```

3.1.4 Deeptracy Core

Deeptracy core is a shared library that has common functionalities used in the rest of the projects. When developing is recommended to install it in your virtualenv in editable mode:

```
$ cd deeptracy-core
$ pip install -e .
```

This will instruct distutils to setup the core project in to development mode

3.1.5 Deeptracy Workers

This project is a Celery project. You can install it with:

```
$ cd deeptracy
$ make install-requirements_dev
```

3.1.6 Deeptracy API

This project is a **'Flask'_** project. You can install it with:

```
$ cd deeptracy-api
$ make install-requirements_dev
```

3.1.7 Dependencies

These distributions will be installed automatically when installing Deeptracy.

- Celery is an asynchronous task queue/job queue based on distributed message passing
- Redis in-memory data structure store used as message broker in celery
- Psycopg PostgreSQL database adapter for Python
- Pluginbase for plugin management
- Docker most tasks are executed inside docker containers
- PyYAML parse yml files

3.1.8 Development dependencies

These distributions will be installed for development and local testing

- Bumpversion control the version numbers in releases.
- Sphinx documentation generation
- Flake8 for linting and code style
- Coverage checks code coverage
- Behave acceptance tests

3.2 Usage

3.2.1 Makefiles & Dotenv

To standardize tasks among repositories, each repository have a `Makefile` that can be used to perform common tasks. By executing `make` in the root of each project you can get a detailed list of tasks that can be performed.

When executing tasks with make, we also provide a `.dot-env` mechanism to have local environment variables for each project. So, the first time you perform any make task, you will be prompted for the required environment variables for that project.

Keep in mind that you can always change the local environment for a project by editing the `.env` file generated in the project root folder.

This is a sample of common tasks that can be performed with make:

```
$ make
clean                remove all build, test, coverage and Python artifacts
test                 run tests quickly with py.test
test-all             run tests on every python version with tox
lint                 check style with flake8
coverage             check code coverage
docs                 generate and shows documentation
run                  launch the application
at_local             run acceptance tests without environemnt. You need to start your␣
↪own environment (for dev)
at_only              run acceptance tests without environemnt, and just features␣
↪marked as @only (for dev)
at                   run acceptance tests in complete docker environment
```

3.2.2 Local environment

You can have a full functional working local environment to do integration or acceptance tests. En the workers and API projects you can find a `docker-compose-yml` file that will launch a postgres and a redis container:

```
$ cd deeptracy
$ docker-compose up
```

Once the database and the broker are in place, now you can launch each project issuing a `make run` on each of them.

3.2.3 Development flow

You should be doing unit test to test the new features. When you are working in **deeptracy** or in **deeptracy-api** is likely you will also need to work in **deeptracy-core**. If you installed the core in *Deeptracy Core* you will see the changes in the core from the other projects as soon as they are made.

Once the new feature is covered and tested with unit tests, you can launch a *Local environment* and run the acceptance tests in the local environment with `make at_local`

3.2.4 Deeptracy Worker

Deeptracy worker has the celery tasks and worker to process them. The actual task flow is as follows:

```
                                                  >- Run analyzer ->
Prepare Scan -> Scan Dependencies -> Start Scan ->>- Run analyzer ->-> Merge Results -
↪> Notify
                                                  >- Run analyzer ->
```

Prepare Scan

This task is the first task in the chain to scan projects. It is responsible of two things:

- Clone the repository
- Ensure the scan has a language stored. If it is not present in the database, try to extract it from the `.deeptracy.yml` if it is present in the repository

Scan Dependencies

After running the actual scan, this task extracts all the dependencies for the project and store them in to the database. The dependency list is compared with the last previous scan to check for differences. If no differences are found in the dependency graph, the scan is aborted.

Start Scan

This task check the language of the scan and launches a task for each plugin available for that language. For each plugin, this task will create a scan analysis in the database and launch the task that will perform the actual scan for that plugin.

The task for the analysis are launched in parallel.

Run Analyzer

This task is the responsible to do the actual vulnerability scan in the source code. It will invoke the corresponding plugin and then return a serialized return with each vulnerability found.

Merge Results

After all the analysis have being made, all results are sent to this task to merge the results to avoid dupplicated results, and store the final vulnerability list in to the database.

If the project has a notification hook, this task spawn the final task to notify the project about the scan.

Notify Results

This task sends a notification to the project about the finished scan and the vulnerabilties that has being found.

3.3 Testing

3.3.1 Unit Tests

For development is recommended to do unit tests to speedup the process (you don't need a full environment), and only do acceptance and integration tests when the feature is ready and tested with unit tests.

> **Warning:** Pipelines has a check on whether the test coverage has a minimum of code covered, so lowering the percentage of lines of code covered by unit tests is not an option. You can check your code coverage with `make coverage`

3.3.2 Acceptance Tests

3.3.3 Code Coverage

CHAPTER 4

Source Code Docs

4.1 deeptracy package

4.1.1 Subpackages

deeptracy.notifications package

Submodules

deeptracy.notifications.slack_webhook_post module

Detailed documentation of Slack Incoming Webhooks: https://api.slack.com/incoming-webhooks

`deeptracy.notifications.slack_webhook_post.`**`notify`**(*webhook_url: str*, *text*)

Module contents

deeptracy.tasks package

Submodules

deeptracy.tasks.base_task module

This module contains base class for all celery task in deeptracy and other common classes used in all tasks

`class` `deeptracy.tasks.base_task.`**`DeeptracyTask`**

Bases: `celery.app.task.Task`

Default class for all task in deeptracy. It has error handling for logging all celery failures in tasks

`on_failure`(*exc*, *task_id*, *args*, *kwargs*, *einfo*)

exception deeptracy.tasks.base_task.**TaskException**
Bases: BaseException

Exception for use in controlled errors inside tasks

deeptracy.tasks.merge_results module

deeptracy.tasks.notify_results module

deeptracy.tasks.prepare_scan module

deeptracy.tasks.prepare_scan.**clone_project**(*base_path: str*, *scan_id: str*, *repo_url: str*, *repo_auth_type: str*) → str

Clone a project repository.

This method handles repository auth if the repo is not public. The repository is going to be cloned in "(base_path)/(scan_id)/sources" folder (it will be created).

To do the clone a docker image is used: *bravissimolabs/alpine-git*

Parameters

- **base_path** – (str) Base path to clone. Should be an absolute path
- **scan_id** – (str) Scan id that triggers the clone. Its going to be created as a folder inside the base_path
- **repo_url** – (str) project repository url to make the git clone
- **repo_auth_type** – (str) if the repo needs any kind of auth

Returns returns the sources path where the repo is cloned

deeptracy.tasks.prepare_scan.**parse_deeptracy_yml**(*source_dir: str*)
Find a .deeptracy.yml file inside source_dir and try to parse it.

If the file is not found, return None

Parameters **source_dir** –

Returns None is the file is not found or cant be parsed, else it returns a dict with the key/values

deeptracy.tasks.prepare_scan.**prepare_path_to_clone_with_local_key**(*scan_path: str*, *repo: str*, *mounted_vol: str*, *source_folder: str*)

Prepare a folder to clone a repository which needs a local private key.

LOCAL_PRIVATE_KEY means we need to copy the local private key present in the host to the folder that is being mounted in to the container that is going to perform the actual repo clone

We prepare a script with all the commands needed to perform the clone, like adding the private key to the container ssh-agent and disabling host-key verification

Parameters

- **scan_path** – (str) path that we need to prepare
- **repo** – (str) project repository to clone

- **mounted_vol** – (str) path to the mounted volume in the container that makes the clone
- **source_folder** – (str) name of the folder to made the actual clone

Returns returns the command to pass to the container that makes the clone

deeptracy.tasks.run_analyzer module

deeptracy.tasks.scan_deps module

deeptracy.tasks.scan_deps.**get_dependencies**(*lang: str*, *sources: str*)
Given a language and a sources path, scan that sources to get a complete list of dependencies.

Each language get the dependencies in a different way, but always inside a container to isolate the sources

deeptracy.tasks.scan_deps.**get_dependencies_for_nodejs**(*sources: str*, *mounted_vol: str*, *docker_volumes: dict*)

deeptracy.tasks.start_scan module

Module contents

4.1.2 Submodules

4.1.3 deeptracy.celery module

4.1.4 deeptracy.config module

4.1.5 deeptracy.plugin_store module

class deeptracy.plugin_store.**DeeptracyPluginStore**
Bases: object

get_all_plugin_paths()

get_plugin(*plugin_id: str*)

load_plugins()

class deeptracy.plugin_store.**deeptracy_plugin**(*lang: typing.Union[str, typing.List[str]]*)
Bases: object

4.1.6 Module contents

Deeptracy Workers Package.

This package contains celery workers and tasks to process the deeptracy flow for scanning projects.

Python Module Index

d

Index