Diffy Documentation

Netflix

Dec 03, 2019

CONTENTS

Diffy is a differencing engine for digital forensics and incident response (DFIR) in the cloud. Collect data across multiple virtual machines and use variations from a baseline, and/or clustering, to scope a incident.

CHAPTER ONE

INSTALLATION

1.1 Quickstart

This guide will step you through setting up a Python-based virtualenv, configuring it correctly, and running your first baseline and difference against an autoscaling group (ASG). This guide assumes you're operating on a freshly-installed Ubuntu 16.04 instance. Commands may differ in your environment.

Clone the repo:

```
$ git clone git@github.com:Netflix-Skunkworks/diffy.git && cd diffy
```

Install a virtualenv there:

```
$ virtualenv venv
```

Activate the virtualenv:

```
$ source venv/bin/activate
```

Install the required "dev" packages into the virtualenv:

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

Install the local Diffy package:

```
$ pip install -e .
```

Invoke the command line client with default options, to create a new functional baseline. In the command below, replace the `<asg>` placeholder with the name of your autoscaling group (a concept particular to AWS):

```
$ diffy new baseline <asg>
```

You'll find a JSON file in your current directory. This file contains the observations collected as the baseline.

Next, run an analysis across all members of that autoscaling group, to locate outliers:

```
$ diffy new analysis <asg>
```

When done, deactivate your virtualenv:

```
$ deactivate
```

1.2 Production

We haven't intended for folks to run Diffy in a production environment. However, if you'd like to do that, you'll have to do a few things first, to ensure that it will run reliably and securely.

1.2.1 Basics

TODO

CHAPTER

TWO

DEVELOPERS

2.1 Contributing

Want to contribute back to Diffy? This page describes the general development flow, our philosophy, the test suite, and issue tracking.

2.1.1 Impostor Syndrome Disclaimer

Before we get into the details: **We want your help. No, really.**

There may be a little voice inside your head that is telling you that you're not ready to be an open source contributor; that your skills aren't nearly good enough to contribute. What could you possibly offer a project like this one?

We assure you – the little voice in your head is wrong. If you can write code at all, you can contribute code to open source. Contributing to open source projects is a fantastic way to advance one's coding skills. Writing perfect code isn't the measure of a good developer (that would disqualify all of us!); it's trying to create something, making mistakes, and learning from those mistakes. That's how we all improve.

We've provided some clear *Contribution Guidelines* that you can read below. The guidelines outline the process that you'll need to follow to get a patch merged. By making expectations and process explicit, we hope it will make it easier for you to contribute.

And you don't just have to write code. You can help out by writing documentation, tests, or even by giving feedback about this work. (And yes, that includes giving feedback about the contribution guidelines.)

(Adrienne Friend came up with this disclaimer language.)

2.1.2 Documentation

If you're looking to help document Diffy, your first step is to get set up with Sphinx, our documentation tool. First you will want to make sure you have a few things on your local system:

- python-dev (if you're on OS X, you already have this)
- pip
- virtualenvwrapper

Once you've got all that, the rest is simple:

```
# If you have a fork, you'll want to clone it instead
git clone git://github.com/Netflix-Skunkworks/diffy.git

# Create a python virtualenv
```

(continues on next page)

(continued from previous page)

```
mkvirtualenv diffy

# Make the magic happen
make dev-docs
```

Running `make dev-docs` will install the basic requirements to get Sphinx running.

Building Documentation

Inside the `docs` directory, you can run `make` to build the documentation. See `make help` for available options and the Sphinx Documentation for more information.

2.1.3 Developing Against HEAD

We try to make it easy to get up and running in a development environment using a git checkout of Diffy. You'll want to make sure you have a few things on your local system first:

- python-dev (if you're on OS X, you already have this)
- pip
- virtualenv (ideally virtualenvwrapper)
- node.js (for npm and building css/javascript)
- (Optional) PostgreSQL

Once you've got all that, the rest is simple:

```
# If you have a fork, you'll want to clone it instead
git clone git://github.com/Netflix-Skunkworks/diffy.git

# Create a python virtualenv
mkvirtualenv diffy
```

2.1.4 Coding Standards

Diffy follows the guidelines laid out in pep8 with a little bit of flexibility on things like line length. We always give way for the Zen of Python. We also use strict mode for JavaScript, enforced by jshint.

You can run all linters with `make lint`, or respectively `lint-python` or `lint-js`.

Spacing

Python: 4 Spaces

JavaScript: 2 Spaces

CSS: 2 Spaces

HTML: 2 Spaces

2.1.5 Running the Test Suite

If you've setup your environment correctly, you can run the entire suite with the following command:

```
pytest
```

You'll notice that the test suite is structured based on where the code lives, and strongly encourages using the mock library to drive more accurate individual tests.

Note: We use py.test for the Python test suite.

2.2 Contribution Guidelines

All patches should be sent as a pull request on GitHub, include tests, and documentation where needed. If you're fixing a bug or making a large change the patch **must** include test coverage.

Uncertain about how to write tests? Take a look at some existing tests that are similar to the code you're changing, and go from there.

You can see a list of open pull requests (pending changes) by visiting https://github.com/Netflix-Skunkworks/diffy/pulls

Pull requests should be against **master** and pass all TravisCI checks.

We use pre-commit hooks to help us all maintain a consistent standard for code. To get started, run:

```
pre-commit install
```

Before submitting code, run these:

```
pre-commit run --all-files
```

2.3 Writing a Plugin

Several interfaces exist for extending Diffy:

- Analysis (diffy.plugins.bases.analysis)
- Collection (diffy.plugins.bases.collection)
- Payload (diffy.plugins.bases.payload)
- Persistence (diffy.plugins.bases.persistence)
- Target (diffy.plugins.bases.target)
- Inventory (diffy.plugins.bases.inventory)

Each interface has its own functions that will need to be defined in order for your plugin to work correctly. See *Plugin Interfaces* for details.

2.3.1 Structure

A plugins layout generally looks like the following:

```
setup.py
diffy_pluginnae/
diffy_pluginname/__init__.py
diffy_pluginname/plugin.py
```

The `__init__.py` file should contain no plugin logic, and at most, a VERSION = 'x.x.x' line. For example, if you want to pull the version using pkg_resources (which is what we recommend), your file might contain:

```
try:
    VERSION = __import__('pkg_resources') \
        .get_distribution(__name__).version
except Exception as e:
    VERSION = 'unknown'
```

Inside of `plugin.py`, you'll declare your Plugin class, inheriting from the parent classes that establish your plugin's functionality:

```
import diffy_pluginname
from diffy.plugins.bases import AnalysisPlugin, PersistencePlugin

class PluginName(AnalysisPlugin):
    title = 'Plugin Name'
    slug = 'pluginname'
    description = 'My awesome plugin!'
    version = diffy_pluginname.VERSION

    author = 'Your Name'
    author_url = 'https://github.com/yourname/diffy_pluginname'

    def widget(self, request, group, **kwargs):
        return "<p>Absolutely useless widget</p>"
```

And you'll register it via `entry_points` in your `setup.py`:

```
setup(
    # ...
    entry_points={
       'diffy.plugins': [
            'pluginname = diffy_pluginname.analysis:PluginName'
        ],
    },
)
```

You can potentially package multiple plugin types in one package, say you want to create a source and destination plugins for the same third-party. To accomplish this simply alias the plugin in entry points to point at multiple plugins within your package:

```
setup(
    # ...
    entry_points={
        'diffy.plugins': [
            'pluginnamesource = diffy_pluginname.plugin:PluginNameSource',
            'pluginnamedestination = diffy_pluginname.plugin:PluginNameDestination'
```

(continues on next page)

(continued from previous page)

```
        ],
    },
)
```

Once your plugin files are in place and the `setup.py` file has been modified, you can load your plugin by reinstalling diffy:

```
(diffy)$ pip install -e .
```

That's it! Users will be able to install your plugin via `pip install <package name>`.

See also:

For more information about python packages see Python Packaging

Plugin Interfaces

In order to use the interfaces all plugins are required to inherit and override unimplemented functions of the parent object.

2.3.2 Analysis

Analysis plugins are used when you are trying to scope or evaluate information across a cluster. They can either process information locally or used an external system (i.e. for ML).

The *AnalysisPlugin* exposes on function:

```
def run(self, items, **kwargs):
    # run analysis on items
```

Diffy will pass all items collected it will additionally pass the optional *baseline* flag if the current configuration is deemed to be a baseline.

2.3.3 Collection

Collection plugins allow you to collect information from multiple hosts. This provides flexibility on how information is collected, depending on the infrastructure available to you.

The CollectionPlugin requires only one function to be implemented:

```
def get(self, targets, incident, command, **kwargs) --> dict:
     """Queries system target.

    :returns command results as dict {
        'command_id': [
            {
                'instance_id': 'i-123343243',
                'status': 'success',
                'collected_at' : 'datetime'
                'stdout': {json osquery result}
            }
            ...
            {additional instances}
        ]
```

(continues on next page)

(continued from previous page)

```
    }
    """
```

The *incident* string is intended to document a permanent identifier for your investigation. You may insert any unique ticketing system identifier (for example, *DFIR-21996*), or comment, here.

2.3.4 Payload

Diffy includes the ability to modify the *payload* for any given command. In general this payload is the dynamic generation of commands sent to the target. For instance if you are simply running a *netstat* payload you may have to actually run a series of commands to generate a JSON output from the *netstat* command.

Here again the incident is passed to be dynamically included into the commands if applicable.

The PayloadPlugin requires only one function to be implemented:

```
def generate(self, incident, **kwargs) --> dict:
    # list of commands to be sent to the target
```

2.3.5 Persistence

Persistence plugins give Diffy to store the outputs of both collection and analysis to location other than memory. This is useful for baseline tasks or persisting data for external analysis tasks.

The PersistencePlugin requires two functions to be implemented:

```
def get(self, key, **kwargs):
    # retrieve from location

def save(self, key, item, **kwargs):
    # save to location
```

2.3.6 Target

Target plugins give Diffy the ability to interact with external systems to resolve targets for commands.

The TargetPlugin class requires one function to be implemented:

```
def get(self, key, **kwargs):
    # fetch targets based on key
```

2.3.7 Inventory

Inventory plugins interact with asset inventory services to pull a list of targets for baselining and analysis.

Inheriting from the InventoryPlugin class requires that you implement a `process` method:

```
def process(self, **kwargs):
    # Process a new set of targets from a desired source.
    #
    # This method should handle the interaction with your desired source,
    # and then send the results to :meth:`diffy_api.core.async_baseline`.
```

(continues on next page)

(continued from previous page)

```
    #
    # If you poll the source regularly, ensure that you
    # only request recently deployed assets.
```

Testing

Diffy provides a basic py.test-based testing framework for extensions.

In a simple project, you'll need to do a few things to get it working:

2.3.8 setup.py

Augment your setup.py to ensure at least the following:

```
setup(
    # ...
    install_requires=[
       'diffy',
    ]
)
```

2.3.9 conftest.py

The `conftest.py` file is our main entry-point for py.test. We need to configure it to load the Diffy pytest configuration:

```
from diffy.tests.conftest import *  # noqa
```

2.3.10 Running Tests

Running tests follows the py.test standard. As long as your test files and methods are named appropriately (`test_filename.py` and `test_function()`) you can simply call out to py.test:

```
$ py.test -v
============================== test session starts ==============================
platform darwin -- Python 2.7.10, pytest-2.8.5, py-1.4.30, pluggy-0.3.1
cachedir: .cache
collected 346 items

diffy/plugins/diffy_acme/tests/test_aws.py::test_ssm PASSED

=========================== 1 passed in 0.35 seconds ============================
```

See also:

Diffy bundles several plugins that use the same interfaces mentioned above.

2.4 Internals

CHAPTER THREE

SECURITY

3.1 Security

We take the security of `diffy` seriously. The following are a set of policies we have adopted to ensure that security issues are addressed in a timely fashion.

3.1.1 Reporting a security issue

We ask that you do not report security issues to our normal GitHub issue tracker.

If you believe you've identified a security issue with `diffy`, please report it to `cloudsecurity@netflix.com`.

Once you've submitted an issue via email, you should receive an acknowledgment within 48 hours, and depending on the action to be taken, you may receive further follow-up emails.

3.1.2 Supported Versions

At any given time, we will provide security support for the master branch as well as the 2 most recent releases.

3.1.3 Disclosure Process

Our process for taking a security issue from private discussion to public disclosure involves multiple steps.

Approximately one week before full public disclosure, we will send advance notification of the issue to a list of people and organizations, primarily composed of operating-system vendors and other distributors of `diffy`. This notification will consist of an email message containing:

- A full description of the issue and the affected versions of `diffy`.
- The steps we will be taking to remedy the issue.
- The patches, if any, that will be applied to `diffy`.
- The date on which the `diffy` team will apply these patches, issue new releases, and publicly disclose the issue.

Simultaneously, the reporter of the issue will receive notification of the date on which we plan to make the issue public.

On the day of disclosure, we will take the following steps:

- Apply the relevant patches to the `diffy` repository. The commit messages for these patches will indicate that they are for security issues, but will not describe the issue in any detail; instead, they will warn of upcoming disclosure.
- Issue the relevant releases.

If a reported issue is believed to be particularly time-sensitive – due to a known exploit in the wild, for example – the time between advance notification and public disclosure may be shortened considerably.

The list of people and organizations who receives advanced notification of security issues is not, and will not, be made public. This list generally consists of high-profile downstream distributors and is entirely at the discretion of the `diffy` team.

CHAPTER

FOUR

DOING A RELEASE

4.1 Doing a release

Doing a release of `diffy` requires a few steps.

4.1.1 Bumping the version number

The next step in doing a release is bumping the version number in the software.

- Update the version number in `diffy/__about__.py`.
- Set the release date in the *Changelog*.
- Do a commit indicating this.
- Send a pull request with this.
- Wait for it to be merged.

4.1.2 Performing the release

The commit that merged the version number bump is now the official release commit for this release. You will need to have `gpg` installed and a `gpg` key in order to do a release. Once this has happened:

- Run `invoke release {version}`.

The release should now be available on PyPI and a tag should be available in the repository.

4.1.3 Verifying the release

You should verify that `pip install diffy` works correctly:

```
>>> import diffy
>>> diffy.__version__
'...'
```

Verify that this is the version you just released.

4.1.4 Post-release tasks

- Update the version number to the next major (e.g. `0.5.dev1`) in `diffy/__about__.py` and
- Add new *Changelog* entry with next version and note that it is under active development
- Send a pull request with these items
- Check for any outstanding code undergoing a deprecation cycle by looking in `diffy.utils` for `DeprecatedIn**` definitions. If any exist open a ticket to increment them for the next release.

CHAPTER FIVE

FAQ

5.1 Frequently Asked Questions

5.1.1 Common Problems

5.1.2 How do I

CHAPTER

SIX

REFERENCE

6.1 Changelog

6.1.1 0.1.0 - *master*

Note: This version is not yet released and is under active development

6.2 License

Diffy is licensed under a three clause APACHE License.

The full license text can be found below (*Diffy License*).

6.2.1 Authors

Diffy was originally written and is maintained by Forest Monsen & Kevin Glisson.

A list of additional contributors can be seen on GitHub.

6.2.2 Diffy License