pyt Documentation

Release 1.0.0

Bruno, Stefan, Kevin, Toughee, chrisgavin, Jakub, cclaus, Florian

Aug 03, 2018

User Documentation

pyt is static analysis tool for detecting security vulnerabilities in Python3 web applications. It uses the ast module to parse Python and then performs a variant of reaching definitions to track taint from source to sink.

The main documentation for the tool is organized into a few sections:

CHAPTER 1

Getting Started

This document will show you how to get started using PyT.

1.1 Install

1. git clone https://github.com/python-security/pyt.git
2. python setup.py install
3. pyt -h

1.2 Usage from Source

Using it like a user `python -m pyt -f example/vulnerable_code/XSS_call.py save -du`

Running the tests `python -m tests`

Running an individual test file `python -m unittest tests.import_test`

Running an individual test `python -m unittest tests.import_test.ImportTest.test_import`

CHAPTER 2

PyT Features

This will serve as a list of all of the features that PyT currently has.

- Detect Command injection
- Detect SQL injection
- Detect XSS
- Detect directory traversal
- Detect open-redirects
- Get a control flow graph
- Get a def-use and/or a use-def chain
- Search GitHub and analyse hits with PyT
- Scan intraprocedural or interprocedural
- A lot of customisation possible

Example usage and output:

CHAPTER 3

Frequently Asked Questions

3.1 Why do you not support Python2?

It's on our near-term roadmap, we're currently trying to iron-out our Python3 support before giving our full attention to Python 2 support.

3.2 How do you deal with path explosion?

Because of the way reaching definitions unions predecessors we don't explode for the entire program we're analysing, just during conditionals i.e. after an if statement is done we're back with dealing with one path.

3.3 Do you do inlining or summaries for inter-procedural analysis?

Inlining, but a long-term goal is to re-write PyT to use summaries instead so that we can e.g. analyze a PR and only spend time analysing the difference. This is probably the last thing on our roadmap.

CHAPTER 4

Support

4.1 Bugs & Support Issues

You can file bug reports on our GitHub issue tracker, and they will be addressed as soon as possible. **Support is a volunteer effort**, and there is no guaranteed response time.

4.2 Reporting Issues

When reporting a bug, please include as much information as possible that will help us solve this issue. This includes:

- The command that you're running
- Target file that produces the issue
- Expected result
- Actual result

CHAPTER 5

The Story

PyT was created by Bruno and Stefan, as their Master's Thesis.

Almost a year later, Kevin was talking about JunkHacker with a friend, the friend asked for resources for learning more about taint tracking, while looking he found the PyT thesis, gave up on JunkHacker and has been contributing since.

A couple of months after that Collin released Focuson, we then started to publicize PyT to the Python and InfoSec communities.

CHAPTER 6

Maturity

It is important to state that PyT is still a very young, immature project. As far as we know, nobody runs PyT on their production code at this point in time.

CHAPTER 7

Feature Roadmap

- Documentation
- Nested Function Calls
- Reduce Pagination False Positives
- 3.5 and 3.6 CFG Features
- Run it on many projects, enumerate all the problems and fix some of them
- Python 2 Support
- Add a tox.ini
- PyPI Packaging

CHAPTER 8

Contributing

You are here to help on PyT? Awesome, feel welcome and read the following sections in order to know what and how to work on something. If you get stuck at any point you can create a ticket on GitHub.

Join our slack group: https://pyt-dev.slack.com/ - to ask for an invite, email mr.thalmann@gmail.com

Please make sure you are welcoming and friendly in all of our spaces.

8.1 Contributing to development

If you want to deep dive and help out with development on PyT, then first set up a development environement according to the virtual env setup guide. After that is done we suggest you have a look at tickets in our issue tracker that are labelled Easy. These are meant to be a great way to get a smooth start and won't put you in front of the most complex parts of the system.

Procedure for adding new features:

- Pitch idea in slack
- Create issue in Github
- Develop the feature in a separate feature-branch * Feature branch names should start with the issues number and is allowed to contain letters and underscores, for instance: 12_add_new_awesome_feature * Remember to write unit tests and docstrings for your code, and if necessary documentation
- Announce finished feature as pull request
- Pull request reviewed by at least 1
- Merge to development branch when ready
- Merge into master when we all agree

CHAPTER 9

Development Team

9.1 Members

- Bruno
- Stefan
- Kevin
- Toughee
- chrisgavin
- Jakub
- cclaus
- Florian
- Your Name Here

Feel free to ask any of us if you have questions or want to join!

9.2 Joining

We try to be pretty flexible with who we allow on the development team. The best path is to send a few pull requests, and follow up to make sure they get merged successfully. You can check out *Contributing* to get more information, and find issues that need to be addressed. After that, feel free to ask for a commit bit.

CHAPTER 10

Related Work

10.1 Related Projects

- **Bandit** Bandit was a tool made by a few people at OpenStack with the same purpose of PyT in mind, the main difference is that Bandit doesn't track the flow of data and PyT does, so it's closer to a grep ish pre-commit hook to e.g. ban urllib2 and open etc. and suggest Advocate and a secure open wrapper instead. The sinks, formatters and UI are where it shines.
- **JunkHacker** Written by Kevin Hock and had an incredibly bad design, it analyzed Python bytecode using equip and tracked taint by depth-first searching through basic block's via a bytecode interpreter heavily adopted from Byterun. It dealth with path-explosion via a crazy buddy system and being that many byecode instructions e.g. exceptions do not work on the basic block level, work arounds were gruesome. The buddy system worked by marking each node that diverged with the node that it's children converged at. This was both less efficient than unioning predecessors like PyT and more complicated. It is not open-source because of how ugly it is.
- **Focuson** Written by Collin Greene of Uber, similar to PyT it uses the ast module but unlike PyT it tracks dataflow using path-insensitive backwards slicing. Path explosion is not a problem because it is path-insensitive, but that causes it to have more false-positives than PyT.
- **PyExZ3** A dynamic symbolic execution framework for Python, potentially useful for taint tracking if it can solve string constraints, which there is experimental support for in a fork. "A novel aspect of the rewrite is to rely solely on Python's operator overloading to accomplish all the interception needed for symbolic execution." Joseph Near did this before them, but it is interesting work nevertheless.
- **DARLAB Work** Has great alias analysis work, by Michael Gorbovitski et al. It would be quite performance intensive to add to a security tool and may or may not be that helpful for reducing false positives, but is quite impressive work regardless.
- **RIPS (PHP)** The latest versions, the useful ones, are closed-source, as the author Johannes Dahse has gone commercial. This is unfortunate and it seems like the most advanced tool in this category as far as we know because it can find second order vulnerabilities. The old unsophisticated open-source version is here.

- **Brakeman (Rails)** Written by Justin Collins, it is written in Ruby and made for Rails. I'm not exactly sure how it works, but it does do something like reaching definitions.
- **Dawnscanner (Ruby)** Written by Paolo Perego, I'm not exactly sure how it works.
- **Joseph Near et al. (Rails)** Joseph has a lot of interesting work I would like to summarize.

10.2 Related Papers

- Schwarzbach static analysis notes The PyT thesis is heavily influenced by these notes, they're a pretty good resource for learning dataflow analysis. Other good resources include Engineering a Compiler, Advanced Compiler Design and Implementation and Data Flow Analysis: Theory and Practice.
- Alias Analysis for Optimization of Dynamic Languages
- **Static Detection of Second Order Vulnerabilities in Web Applications** A simple intuitive idea, but complex to implement. Unlike PyT they use summaries instead of inlining, summaries are sort of required to implement the idea, unless you wrote results somewhere and ran the tool again with those results in a dirty hack. The main hard-parts with implementing this idea with PyT will be (1) re-writing to use summaries, (2) writing code that deals with this part of the paper "SQL has different syntactical forms of writing data to a table. Listing 1 shows three different ways to perform the same query". Aside from the examples given in the paper, some other examples of multi-step exploits are as follows. Tracking from bad RNG to store in location A to HTTP response, then seeing where a taint value is checked against location A.

  In my opinion, the best ROI in the Python world would be to implement this for the Django ORM or SQLAlchemy since they seem to be the most widely used.
- Finding Security Bugs in Web Applications using a Catalog of Access Control Patterns
- Derailer Interactive Security Analysis for Web Applications
- **Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries** There might be 3 ways of handling blackbox calls between source and sink, to basically answer the questions that a proper summary does, e.g. if argument A is tainted, does this call return a tainted value? This can be dealth with via (1) hard-coded mapping, (2) pip install, see if Python code or, (3) possibly this paper. I suspect long-term, some combination of 1 and 2 will be done with PyT. If we just ask the user, "Hey, does this call propagate taint?" and we remember the answer, it would be easy enough for the user to use the tool.

CHAPTER 11

Past Evaluations

11.1 November 16-17th, 2017

So back in the day Bruno had made a large CSV file of open-source Flask repositories, 547 unique ones to be exact, 66 had problems downloading (maybe removed between now and then) so I cloned 481 unique open-source repos.[1]

I decided, to stay focused during the hackathon, and complete my goal in time, that I would only look for open-redirects. Open-redirects are when you can send somebody a link like `https://chase.com?next=http://evil.com` or `https://chase.com?next=%68%74%74%70%73%3a%2f%2f%77%77%77%2e%65%76%69%6c%2e%63%6f%6d` and evil.com has a phishing page on it. The reason I chose open-redirects as my one bug class is because in my experience, there are less secondary nodes involved. e.g.

```
@app.route('/login/authorized')
@facebook.authorized_handler
def facebook_authorized(resp):
    next_url = request.args.get('next') or url_for('new_paste')
    if resp is None:
        flash('You denied the login')
        return redirect(next_url)
    # ...
```

By 'secondary nodes', I mean the arg typically is retrieved and then placed in `redirect()`, it isn't passed into another function, or used in an assignment somewhere else. This makes them somewhat easier to find statically, as there are less things that can go wrong.

In order to evaluate the tool, I needed to know which repositories actually had vulnerabilities in them, so I put my junk hacking skills to work and made 2 regular expressions. `".*redirect\([a-zA-Z0-9_]+\).*"` and `".*redirect\(request.*\).*"`. The first one detects any redirect call with a variable as the only argument, and the second detects any redirect call with request.anything as the first argument. These aren't perfect, but have a good chance of detecting 90 something percent of possibly vulnerable calls. (The CSV file somehow had a controller file associated with each repo, and each regex was run on every line.)

[1] There are some commits in the CSV file that removed a few repos that caused issues before, but those commits were made long ago, just wanted to note the stats aren't 100% impartial.

This narrowed it down to 20 repositories, around ~4.1 percent. 17 repos matched the first regex, and 3 matched the second.[2]

The great: 4 true positives were reported, although 3 of them are in `@twitter.authorized_handler` or `@facebook.authorized_handler` controllers (used in OAuth.) The only notable one is noted first, as it is in code written by the creator of Flask.

- mitsuhiko/flask-pastebin 5 vulns reported, 1, 2, 3: really false: reported as unknown (GitHub issues incoming, but partially due to us not checking the "Does this return a tainted value with tainted args?" mapping for `url_for`) 4,5: true: true
- ashleymcnamara/social_project_flask 2 vulns: really true, reported as true
- fubuki/python-bookmark-service 1 vuln: really true, reported as true
- GandalfTheGandalf/twitter 2 vulns: really true, reported as true

The good: 7 didn't have any vulnerabilities, and we didn't report any.

- gene1wood/flaskoktaapp
- lepture/flask-oauthlib
- sijinjoseph/multunus-puzzle
- AuthentiqID/examples-flask
- honestappalachia/honest_site
- jpf/okta-pysaml2-example
- ciaron/pandaflask_old

The bad: 4 had no real vulnerabilities, but had one or two false positives.

- billyfung/flask_shortener 2 false positives (Lazy mistake of making `.get` a source instead of `request.args.get` etc., so `redis.get` was used as a source.)
- ZAGJAB/Flask_OAuth2 1 false positive (Customization is needed for open-redirects, to eliminate all false-positives, because if something tainted is used in string formatting, it typically needs to be at the very beginning of the string to be a vulnerability.)
- amehta/Flaskly 1 false positive (GitHub issue incoming.)
- mskog/cheapskate 1 false positive (GitHub issue incoming.)

The ugly: 2 had one false-negative.

- oakie/oauth-flask-template (GitHub issue incoming.)
- cyberved/simple-web-proxy (GitHub issue incoming.)

The fatal: 3 crashed PyT, but by sheer luck, didn't have any open-redirect vulnerabilities in them, I looked. 2 or 3 of them were caused by a PR I merged last weekend.

- commandknight/cs125-fooddy-flask (GitHub issue incoming.)
- bear/python-indieweb (GitHub issue incoming.)
- ubbochum/hb2_flask (GitHub issue incoming.)

In closing, these results seem okay to me, because once the GitHub issues are made for these issues and fixed we'll be in a great place. This evaluation certainly didn't find all the bugs, but probably most of them. The next time I evaluate the tool, I think I'll look for command injection vulnerabilities or SSRF. I think the only takeaway is that for evaluating a tool like this you should use regexes to narrow down what you spend your time analyzing. I'll make a PR

[2] This isn't exactly true, if something matched the 2nd regex I removed `".*redirect\(request\.url\).*"` and `".*redirect\(request\.referrer\).*"`.

in the PyT repo to show what I customized to just find open-redirects, but it was mostly trimming down the sink list to just `redirect` and removing `form` from the source list.

11.2 Around or before May 2016

During the writing of the original thesis PyT was run on 6 or 7 open-source projects and no vulnerabilities were found. I think the sample size was too small.

CHAPTER 12

Virtual env setup guide

Create a directory to hold the virtual env and project

`mkdir ~/a_folder`

`cd ~/a_folder`

Clone the project into the directory

`git clone https://github.com/python-security/pyt.git`

Create the virtual environment

`python3 -m venv ~/a_folder/`

Check that you have the right versions

`python --version` sample output `Python 3.6.0`

`pip --version` sample output `pip 9.0.1 from /Users/kevinhock/a_folder/lib/python3.6/site-packages (python 3.6)`

Change to project directory

`cd pyt`

Install dependencies

`pip install -r requirements.txt`

`pip list` sample output

```
gitdb (0.6.4)
GitPython (2.0.8)
graphviz (0.4.10)
pip (9.0.1)
requests (2.10.0)
setuptools (28.8.0)
smmap (0.9.0)
```

In the future, just type `source ~/a_folder/bin/activate` to start developing.

CHAPTER 13

Testing

Before contributing to PyT, make sure your patch passes our test suite by running *python -m tests*.

13.1 Continuous Integration

The PyT test suite is exercised by Travis CI on every push to our repo at GitHub. You can check out the current build status: https://travis-ci.org/python-security/pyt

CHAPTER 14

Building and Contributing to Documentation

As one might expect, the documentation for PyT is built using Sphinx and hosted on Read the Docs. The docs are kept in the `docs/` directory at the top of the source tree.

You can build the docs by installing `Sphinx` and running:

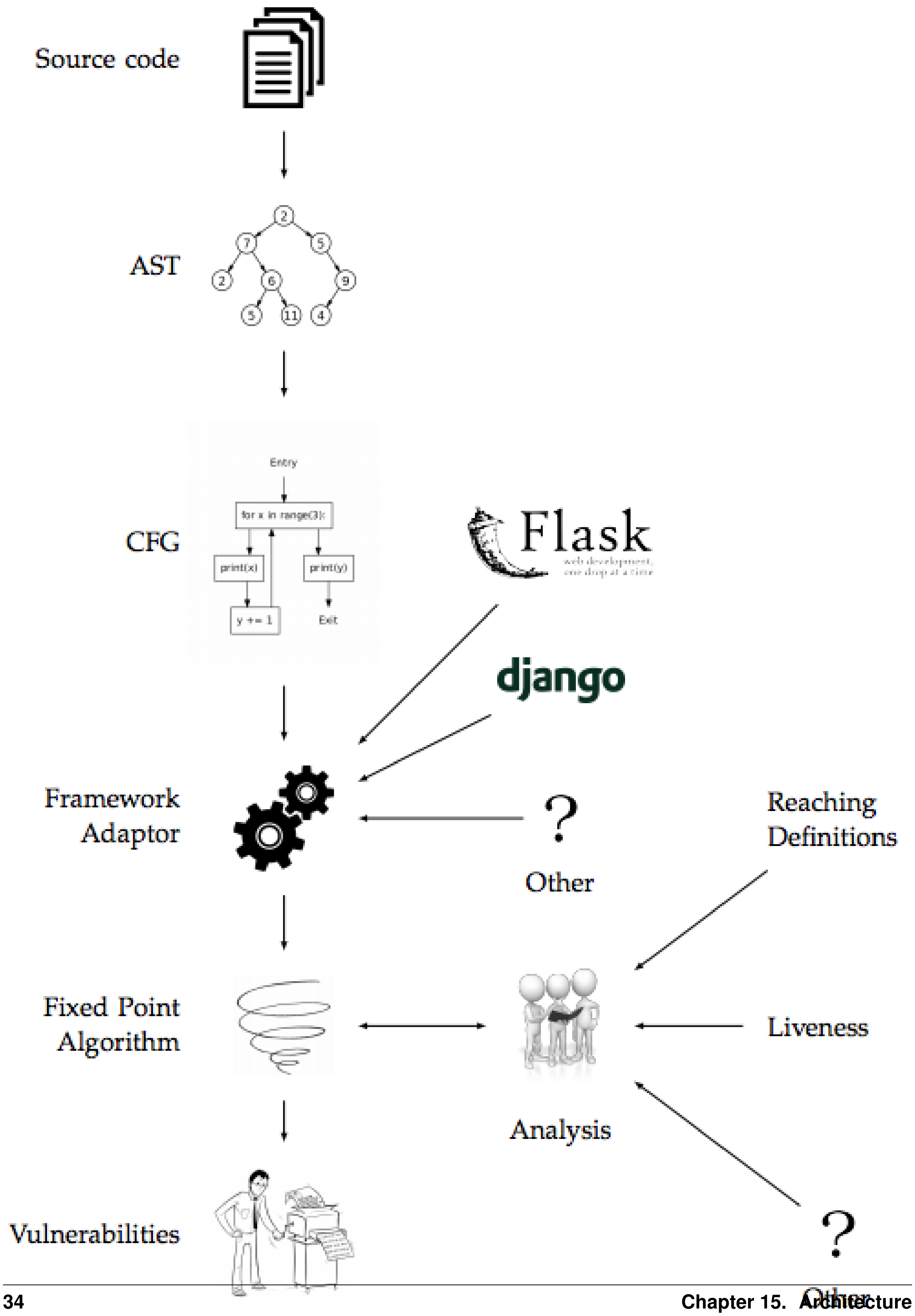
```
# in the docs directory
make html
```

Let us know if you have any questions or want to contribute to the documentation.

CHAPTER 15

Architecture

A great overview from the original thesis is below:


Source code
AST
2
7
5
2
6
9
5
11
4
CFG
Entry
for x in range(3):
print(x)
print(y)
y += 1
Exit
Flask
web development, one drop at a time
django
Framework Adaptor
?
Other
Reaching Definitions
Fixed Point Algorithm
Liveness
Analysis
Vulnerabilities
?
Other

1. Source code to AST
2. AST to CFG
3. CFG entered into a Framework Adaptor
4. Running a fixpoint algorithm on the result of 3
5. Spit out all the vulnerabilities.

I'll go through each of these steps in depth and walk through where in the code they happen.

15.1 Source code to AST

This is by far the easiest step, as it is done for us by the ast module, the only place where we perform parsing is the `generate_ast` function in the ast_helper.py file, where we just write `ast.parse(f.read())` on a file. The result is a tree of objects whose classes all inherit from ast.AST.

15.2 AST to CFG

This is mostly performed by a class that inherits from ast.NodeVisitor, named Visitor in base_cfg.py.

CHAPTER 16

How It Works

- **What is an AST?** The wikipedia page of abstract syntax tree is pretty clear. All the different kinds of nodes in Python are documented on Green Tree Snakes,
- **What is the best beginner friendly resource to learning about dataflow analysis?** Probably just Engineering a Compiler. All you need to know at a minimum to understand PyT is reaching definitions.
- **What is reaching definitions?** The wikipedia page of reaching definitions analysis is pretty clear. It is a type of dataflow analysis.
- What is liveness analysis?
- **What does fixed point mean?** Iterate and perform this dataflow analysis until nothing changes. The until nothing changes part is what is known as a fixed point.
- **How does fixpointmethod affect constraint table?** It depends on the dataflow analysis, but
- **What is a lattice?** A lattice is just a partially ordered set that sounds fancy and looks pretty in papers. For example, in the case of reaching definitions, the lattice is made up of all the assignments in the program.
- **What design patterns are used in the PyT codebase?** The template pattern is used when deciding what analysis to use. The visitor pattern is used throughout the codebase, when visiting the different nodes of the AST.