Cork|Sec 73

Introduction to static code analysis for security

WITH BANDIT FOR PYTHON

Yassine Ilmi

Some terminology...

SCA - Static Code Analysis

SAST - Static Application Security Testing

DAST - Dynamic Application Security Testing

IAST - Interactive Application Security Testing

But...also

SCA - Software Composition Analysis

A few approaches to analyzing code

Considering a large project

Dynamic Code Analysis

Cost depending on type of tool
- Zap/Burp (Medium)
- AFL (Expensive)

Full coverage difficult
- Still have to do some work

Running it can take some time

Examples:
- Fuzzing or instrumented fuzzing
- Tests manual/automated

Manual Review

Human review takes time

Tedious, full coverage difficult

Requires knowledge of security issues

Inconsistent methodology and knowledge between reviewers

Static Code Analysis

Cheap to run

Better coverage

Usually Fast

Quality vary between tools rules and scanning engine

Consistency across scans

Examples:
- As simple as a grep
- Style Checkers and other linters

SCA is not only about security

- Definition: a tool that analyzing a program's source code in a static way (not actively executing)
  - Performed directly on the source, but could also use generated object or bytecode
- Compilers, Interpreters, Transpilers all performing static code analysis at some point
  - Syntactical analysis
  - Lexical analysis
  - Others checks too (unused variables, inconsistency, etc…)
- Style checking is also a form of static code analysis
  - Indentation, spacing, line length, function documentation, etc…
- Basically any coding practice enforced by checking the code without executing it
  - e.g. grep using different level of regexp could also be considered SCA

Static Code Analysis for Security

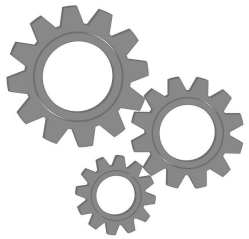
CONCEPTS

Overall workflow

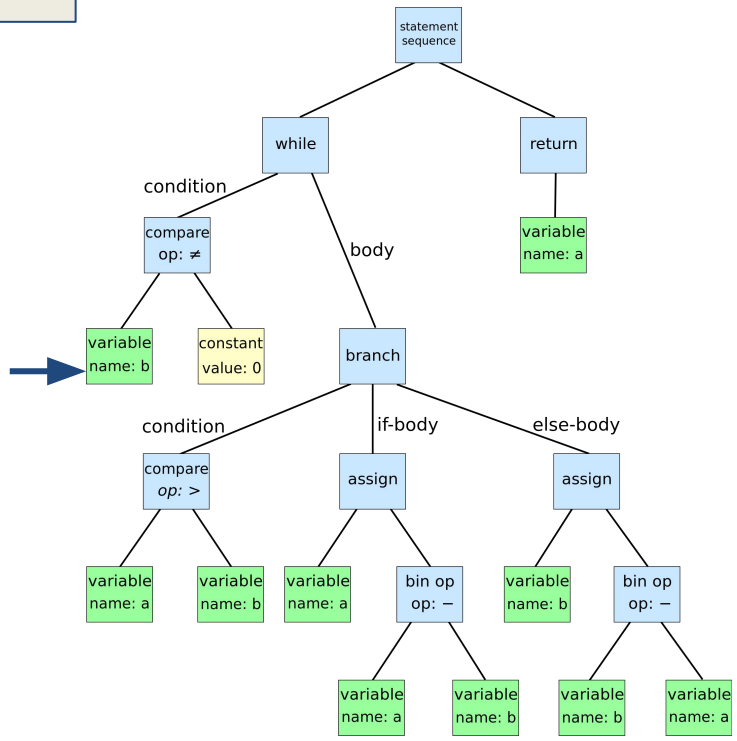
Source code

```
def function(untrusted input):
    do_stuff(input)
    ....
```

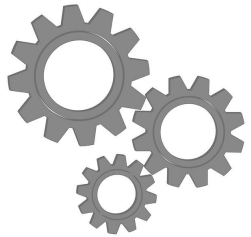
Translation



Model (AST for bandit)

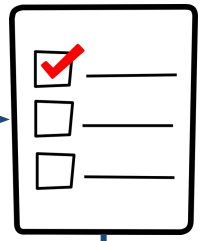


Analysis



Rules

Results



Review

About Bandit

Bandit was first created for the OpenStack project developed in Python.

Its objective was to find common security issues in Python code. To do so, it builds an Abstract Syntax Tree from the source code and runs plugins against the model.

Bandit is now maintained by the Python Code Quality Authority - PyCQA.

But, how does it work exactly?

If we take Bandit for example

- It uses the Python standard ast module to build an abstract syntax tree
  - Builds an Abstract syntax tree representation for Bandit
  - Provides the control flow graph used to identify all possible execution flow
- Supports plugins for more accurate and specific analysis
  - A plugin can go through the AST and perform different type of checks
  - Some plugins will perform simple checks without considering safe usage creating FP
- Has a list of api's that are "blacklisted"
  - The plugin handling the blacklisted APIs identification is B001
  - Blacklisted because they are known to be dangerous
    - Pickle which could lead to Remote Code Execution (RCE)
    - PyYAML with the unsafe default yaml.load()

Some examples

AND WHY WE NEED THE BEST COVERAGE POSSIBLE

Reflected XSS

```
from flask import Flask, request, make_response
app = Flask(__name__)

@app.route('/XSS_param', methods =['GET'])
def XSS1():
    param = request.args.get('param', 'not set')

    other_var = param + ''

    html = open('templates/XSS_param.html').read()
    resp = make_response(html.replace('{{ param }}', other_var))
    return resp

if __name__ == '__main__':
    app.run(debug= True)
```

Command Injection

```
import os
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/menu', methods=['GET'])
def menu():
    param = request.args.get('param', 'not set')
    command = 'echo ' + param + ' >> ' + 'menu.txt'
    os.system(command)
    with open('menu.txt', 'r') as f:
        menu_ctx = f.read()
    return render_template('command_injection.html', menu=menu_ctx)

if __name__ == '__main__':
    app.run(debug=True)
```

SQL Injection

```
@app.route('/raw')
def index():
    param = request.args.get('param', 'not set')
    result = db.engine.execute(param)
    print(User.query.all(), file=sys.stderr)
    return 'Result is displayed in console.'

@app.route('/filtering')
def filtering():
    param = request.args.get('param', 'not set')
    Session = sessionmaker(bind=db.engine)
    session = Session()
    result = session.query(User).filter("username={}".format(param))
    for value in result:
        print(value.username, value.email)
    return 'Result is displayed in console.'
```

This example is truncated

Arbitrary file overwrite

```
import tarfile
    tar = tarfile.open("sample.tar.gz")
    tar.extractall()
    tar.close()
```

A description of this issue is given in the python documentation:

- https://docs.python.org/3.7/library/tarfile.html#tarfile.TarFile.extractall

Also, if you want to create an archive that could exploit this issue, check:

- https://github.com/ptoomey3/evilarc.git

Trusting all certificates (MITM)

```
import sys, requests

def get_data(username, password):
    auth = requests.auth.HTTPBasicAuth(username,password)
    session = requests.Session()
    session.verify = True
    response = session.get('https://www.google.com/authenticate' , verify=False, auth=auth)
    response.raise_for_status()
    return response.content

if __name__ == "__main__":
    if len(sys.argv) == 3:
        data = get_data(sys.argv[1], sys.argv[2])
        print(data)
    else:
        print("Invalid number of arguments, require username and password" )
```

Hands on with Bandit

FOR PYTHON PROJECTS

Bandit Setup

- Get Bandit
  - pip/pip3 install bandit
- Test projects
  - https://github.com/ytdl-org/youtube-dl
  - https://github.com/piqueserver/piqueserver/
- Run bandit with different security levels and excluding tests
  - Issues severity: high with -lll, medium with -ll, low with -l
  - Recursive: -r
  - Exclude: -x path/, other_path/
  - Output an html report: -f html -o myreport.html
  - Other interesting switches, for example to aggregate issues by criteria
- Easy to integrate to your pipelines, as a git hook, any ci config (travis, gitlab,...)
  - Examples are available on bandit's repository

Running bandit

- youtube-dl
  - git clone https://github.com/ytdl-org/youtube-dl.git
  - cd youtube-dl
  - bandit bandit . -r -x test/ -l -f html -o low-severity-youtube-dl.html

- piqueserver
  - git clone https://github.com/piqueserver/piqueserver.git
  - cd piqueserver
  - git checkout -b v0.1.1 (has the following bug)
  - bandit . -r -x test/ -lll -f html -o high-severity-project1.html

High vs Low? - youtube-dl

Example of insecure/deprecated TLS/SSL protocols

ssl_with_bad_version: ssl.wrap_socket call with insecure SSL/TLS protocol version identified, security issue.
Test ID: B502
Severity: HIGH
Confidence: HIGH
File: ./youtube_dl/utils.py
More info: https://bandit.readthedocs.io/en/latest/plugins/b502_ssl_with_bad_version.html

```
2517                            sock, self.key_file, self.cert_file,
2518                            ssl_version=ssl.PROTOCOL_TLSv1)
2519                    else:
2520                        self.sock = sock
2521                hc.connect = functools.partial(_hc_connect, hc)
```

ssl_with_no_version: ssl.wrap_socket call with no SSL/TLS protocol version specified, the default SSLv23 could be insecure, possible security issue.
Test ID: B504
Severity: LOW
Confidence: MEDIUM
File: ./youtube_dl/utils.py
More info: https://bandit.readthedocs.io/en/latest/plugins/b504_ssl_with_no_version.html

```
2696                    else:
2697                        self.sock = ssl.wrap_socket(self.sock)
2698
```

Looking at the documentation

B502: ssl_with_bad_version - https://bandit.readthedocs.io/en/latest/plugins/b502_ssl_with_bad_version.html

bandit.plugins.insecure_ssl_tls.ssl_with_bad_version(*context*, *config*)[source]

B502: Test for SSL use with bad version used

Several highly publicized exploitable flaws have been discovered in all versions of SSL and early versions of TLS. It is strongly recommended that use of the following known broken protocol versions be avoided:

- SSL v2
- SSL v3
- TLS v1
- TLS v1.1

This plugin test scans for calls to Python methods with parameters that indicate the used broken SSL/TLS protocol versions. Currently, detection supports methods using Python's native SSL/TLS support and the pyOpenSSL module. A HIGH severity warning will be reported whenever known broken protocol versions are detected.

B504: ssl_with_no_version - https://bandit.readthedocs.io/en/latest/plugins/b504_ssl_with_no_version.html

bandit.plugins.insecure_ssl_tls.ssl_with_no_version(*context*)[source]

B504: Test for SSL use with no version specified

This plugin is part of a family of tests that detect the use of known bad versions of SSL/TLS, please see ../plugins/ssl_with_bad_version for a complete discussion. Specifically, This plugin test scans for specific methods in Python's native SSL/TLS support and the pyOpenSSL module that configure the version of SSL/TLS protocol to use. These methods are known to provide default value that maximize compatibility, but permit use of the aforementioned broken protocol versions. A LOW severity warning will be reported whenever this is detected.

Missed stuff - Trusting all certificates - 1/2

Our previous example is not flagged by bandit, but this one will.

In this case he plugin only checks for the use of *requests.get()* and not for *requests.session.get()*.

```
import sys, requests

def get_data(username, password):
    auth = requests.auth.HTTPBasicAuth(username,password)
    response = request.get('https://www.google.com/authenticate', verify=False, auth=auth)
    response.raise_for_status()
    return response.content

if __name__ == "__main__":
    if len(sys.argv) == 3:
        data = get_data(sys.argv[1], sys.argv[2])
        print(data)
    else:
        print("Invalid number of arguments, require username and password)"
```

Missed stuff - Trusting all certificates - 2/2

```
import sys, requests

def get_data(username, password):
    auth = requests.auth.HTTPBasicAuth(username,password)
    session = requests.Session()
    session.verify = True
    response = session.get('https://www.google.com/authenticate', verify=False, auth=auth)
    response.raise_for_status()
    return response.content

if __name__ == "__main__":
    if len(sys.argv) == 3:
        data = get_data(sys.argv[1], sys.argv[2])
        print(data)
    else:
        print("Invalid number of arguments, require username and password" )
```

Missed stuff - Arbitrary file overwrite

This for example is not flagged by bandit also because the extractall function is not listed, or it's use is not checked by a specific plugin.

```
import tarfile
    tar = tarfile.open("evilfile.tgz")
    tar.extractall()
    tar.close()
```

In our case we checked out the tag that had the tarfile issue:

- https://github.com/piqueserver/piqueserver/commit/3b995250b93755e4c10ab07de7d21a68cdfeb3e4

Review of some limitations

Careful not to get a false sense of security! No silver bullet...

- Low severity issues are not always low
  - Inconsistencies between SSL/TLS protocols warning
  - All issues need to be reviewed
- Even with a good coverage, results depends on the quality of its rules and plugins
  - Plugin crypto_request_no_cert_validation.py not catching *requests.Session().get()*
  - tarfile.extractall() is not blacklisted or verified by a plugin
- But, even if we had a plugin for tarfile.extractall(), the plugin would have to account for safe usage of the function (members=sanitizing_function())
  - Plugins/Rules must be well defined to limit as much as possible false positives

Roll out static code analysis

Considerations

- Look for volunteers to own the SCA, better if curious or concerned about security
  - Start as early as possible in the project life, fix and learn as you go
  - May require some investment to reduce false positives
  - Provide least, some time to learn the tool, at best some tool + security training
- Define clearly your process, at which level your security quality gates are
- For large teams and projects
  - Will likely require some culture change
  - + if the SCA expert has a good knowledge/experience of the project

Takeaways

- Start early, make it part of pre-commit checks, pre-receive, merge, etc…
- Review the tools available for your projects
  - Language specific (projects with pieces written in different languages)
  - Focus specific: Security, bad practices, etc…
- In some cases, using multiple tools can provide broader coverage
  - Tools with different rules or engines

Resources

Security resources

https://www.owasp.org/

Examples of safe vs unsafe implementations:

- https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards
- Vulnerable code examples: https://samate.nist.gov/SRD/index.php
  - Searchable, select status accepted only: https://samate.nist.gov/SRD/search.php

Other scanners

- TypeScript: tslint with microsoft rules for security - https://github.com/microsoft/tslint-microsoft-contrib
- Golang Linter - https://github.com/golang/lint
- Spotbugs (successor of findbugs): https://spotbugs.github.io/
  - Security plugin for spotbugs: https://find-sec-bugs.github.io/
- More (.NET, …): https://www.owasp.org/index.php/Static_Code_Analysis

Some tools will provide a CVSSv3 score, more info:

- Calculator: https://www.first.org/cvss/calculator/3.0
- Specifications: https://www.first.org/cvss/specification-document

Thank you

https://www.github.com/yilmi