# Contributing to Bandit

Thanks for considering to take part in the improvement of the Bandit project. Contributions are always welcome!

Here are guidelines and rules that can be helpful if you plan to want to get involved in the project.

#### Table Of Contents

[Code of Conduct](#code-of-conduct)

[How Can I Contribute?](#how-can-i-contribute)

\* [Reporting Bugs](#reporting-bugs)

\* [Suggesting Enhancements](#suggesting-enhancements)

\* [Your First Code Contribution](#your-first-code-contribution)

\* [Pull Requests](#pull-requests)

\* [Commit Message Guidelines](#commit-message-guidelines)

\* [Squash Commits](#squash-commits)

\* [Things You Should Know Before Getting Started](#things-you-should-know-before-getting-started)

\* [Vulnerability Tests](#vulnerability-tests)

\* [Writing Tests](#writing-tests)

\* [Extending Bandit](#extending-bandit)

## Code of Conduct

Everyone who participates in this project is governed by the PyCQA [Code of Conduct](https://github.com/PyCQA/bandit/blob/master/CODE\_OF\_CONDUCT.md#contributor-covenant-code-of-conduct).

## Reporting Bugs

If you encounter a bug, please let us know about it. See the guide here [GitHub issues](https://guides.github.com/features/issues/).

\*\*Before submitting a new issue\*\* you might want to check for an [existing issue](https://github.com/PyCQA/bandit/issues) to know if there is already a reported issue. If an issue is already open please feel free

to add a comment to the existing issue instead of creating a new one.

### Submitting your first issue

We encourage using the issue template to improve quality of reported issues.

Navigate to the issues tab and select `New issue`, then select the \*\*Bug report\*\* template and fill out the form.

To submit a good bug report keep in mind to:

\* Use a descriptive title so other people can understand what the issue is about.

\* Be specific about the details, for example, what command did you use, what version of Bandit did you use, and in what environment you observed the bug (CI or development).

## Suggesting Enhancements

If you want to suggest an enhancement, open a new issue and use the \*\*Feature request\*\* template.

\*\*Before submitting an enhancement\*\* please check for existing [feature requests](https://github.com/PyCQA/bandit/issues?q=is%3Aopen+is%3Aissue+label%3Aenhancement).

Useful things to point out in your feature request:

\* Explain your feature request in a way that everyone can understand

\* Please try to explain how this feature will improve the Bandit project

## Your First Code Contribution

You can start contributing to Bandit project by picking [bug issues](https://github.com/PyCQA/bandit/issues?q=is%3Aopen+is%3Aissue+label%3Abug)

These issues can be easier to resolve rather than a feature request and can get you up and running with the code base.

## Pull Requests

The best way to get started with Bandit is to grab the source:

Fork the repository into one with your username

```shell script

git clone https://github.com/<your username>/bandit.git

```

Create you own branch to start writing code:

```shell script

git checkout -b mybranch

git push origin mybranch

```

You can test any changes with tox:

```shell script

pip install tox

tox -e pep8

tox -e py27

tox -e py35

tox -e docs

tox -e cover

```

If everything is done, proceed with [opening a new pull request](https://help.github.com/en/desktop/contributing-to-projects/creating-a-pull-request)

### Commit Message Guidelines

We follow the commit formatting recommendations found on [Chris Beams' How to Write a Git Commit Message article](https://chris.beams.io/posts/git-commit/).

Well formed commit messages not only help reviewers understand the nature of

the Pull Request, but also assists the release process where commit messages

are used to generate release notes.

A good example of a commit message would be as follows:

```

Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72

characters or so. In some contexts, the first line is treated as the

subject of the commit and the rest of the text as the body. The

blank line separating the summary from the body is critical (unless

you omit the body entirely); various tools like `log`, `shortlog`

and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you

are making this change as opposed to how (the code explains that).

Are there side effects or other unintuitive consequences of this

change? Here's the place to explain them.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded

by a single space, with blank lines in between, but conventions

vary here

If you use an issue tracker, put references to them at the bottom,

like this:

Resolves: #123

See also: #456, #789

```

Note the `Resolves #123` tag, this references the issue raised and allows us to

ensure issues are associated and closed when a pull request is merged.

Please refer to [the github help page on message types](https://help.github.com/articles/closing-issues-using-keywords/)

for a complete list of issue references.

### Squash Commits

Should your pull request consist of more than one commit (perhaps due to

a change being requested during the review cycle), please perform a git squash

once a reviewer has approved your pull request.

A squash can be performed as follows. Let's say you have the following commits:

initial commit

second commit

final commit

Run the command below with the number set to the total commits you wish to

squash (in our case 3 commits):

git rebase -i HEAD~3

You default text editor will then open up and you will see the following::

pick eb36612 initial commit

pick 9ac8968 second commit

pick a760569 final commit

# Rebase eb1429f..a760569 onto eb1429f (3 commands)

We want to rebase on top of our first commit, so we change the other two commits

to `squash`:

pick eb36612 initial commit

squash 9ac8968 second commit

squash a760569 final commit

After this, should you wish to update your commit message to better summarise

all of your pull request, run:

git commit --amend

You will then need to force push (assuming your initial commit(s) were posted

to github):

git push origin your-branch --force

## Things You Should Know Before Getting Started

### Vulnerability Tests

Vulnerability tests or "plugins" are defined in files in the plugins directory.

Tests are written in Python and are autodiscovered from the plugins directory.

Each test can examine one or more type of Python statements. Tests are marked

with the types of Python statements they examine (for example: function call,

string, import, etc).

Tests are executed by the ``BanditNodeVisitor`` object as it visits each node

in the AST.

Test results are managed in the ``Manager`` and aggregated for

output at the completion of a test run through the method `output\_result` from ``Manager`` instance.

### Writing Tests

To write a test:

- Identify a vulnerability to build a test for, and create a new file in

examples/ that contains one or more cases of that vulnerability.

- Consider the vulnerability you're testing for, mark the function with one

or more of the appropriate decorators:

- @checks('Call')

- @checks('Import', 'ImportFrom')

- @checks('Str')

- Create a new Python source file to contain your test, you can reference

existing tests for examples.

- The function that you create should take a parameter "context" which is

an instance of the context class you can query for information about the

current element being examined. You can also get the raw AST node for

more advanced use cases. Please see the context.py file for more.

- Extend your Bandit configuration file as needed to support your new test.

- Execute Bandit against the test file you defined in examples/ and ensure

that it detects the vulnerability. Consider variations on how this

vulnerability might present itself and extend the example file and the test

function accordingly.

### Extending Bandit

Bandit allows users to write and register extensions for checks and formatters.

Bandit will load plugins from two entry-points:

- `bandit.formatters`

- `bandit.plugins`

Formatters need to accept 5 things:

- `manager`: an instance of `bandit manager`

- `fileobj`: the output file object, which may be sys.stdout

- `sev\_level` : Filtering severity level

- `conf\_level`: Filtering confidence level

- `lines=-1`: number of lines to report

Plugins tend to take advantage of the `bandit.checks` decorator which allows

the author to register a check for a particular type of AST node. For example

::

@bandit.checks('Call')

def prohibit\_unsafe\_deserialization(context):

if 'unsafe\_load' in context.call\_function\_name\_qual:

return bandit.Issue(

severity=bandit.HIGH,

confidence=bandit.HIGH,

text="Unsafe deserialization detected."

)

To register your plugin, you have two options:

1. If you're using setuptools directly, add something like the following to

your ``setup`` call::

# If you have an imaginary bson formatter in the bandit\_bson module

# and a function called `formatter`.

entry\_points={'bandit.formatters': ['bson = bandit\_bson:formatter']}

# Or a check for using mako templates in bandit\_mako that

entry\_points={'bandit.plugins': ['mako = bandit\_mako']}

2. If you're using pbr, add something like the following to your `setup.cfg`

file::

[entry\_points]

bandit.formatters =

bson = bandit\_bson:formatter

bandit.plugins =

mako = bandit\_mako