Bandit Documentation

Release

OpenStack Foundation

February 19, 2022

Contents

Bandit is a tool designed to find common security issues in Python code. To do this, Bandit processes each file, builds an AST from it, and runs appropriate plugins against the AST nodes. Once Bandit has finished scanning all the files, it generates a report.

This documentation is generated by the Sphinx toolkit and lives in the source tree.

CHAPTER 1

Getting Started

1.1 Configuration

Bandit is designed to be configurable and cover a wide range of needs, it may be used as either a local developer utility or as part of a full CI/CD pipeline. To provide for these various usage scenarios bandit can be configured via a YAML file. This file is completely optional and in many cases not needed, it may be specified on the command line by using *-c*.

A bandit configuration file may choose the specific test plugins to run and override the default configurations of those tests. An example config might look like the following:

```
### profile may optionally select or skip tests

# (optional) list included tests here:
tests: ['B201', 'B301']

# (optional) list skipped tests here:
skips: ['B101', 'B601']

### override settings - used to set settings for plugins to non-default values

any_other_function_with_shell_equals_true:
  no_shell: [os.execl, os.execle, os.execlp, os.execlpe, os.execv, os.execve,
    os.execvp, os.execvpe, os.spawnl, os.spawnle, os.spawnlp, os.spawnlpe,
    os.spawnv, os.spawnve, os.spawnvp, os.spawnvpe, os.startfile]
  shell: [os.system, os.popen, os.popen2, os.popen3, os.popen4,
    popen2.popen2, popen2.popen3, popen2.popen4, popen2.Popen3,
    popen2.Popen4, commands.getoutput,  commands.getstatusoutput]
  subprocess: [subprocess.Popen, subprocess.call, subprocess.check_call,
    subprocess.check_output,
    utils.execute, utils.execute_with_timeout]
```

If you require several sets of tests for specific tasks, then you should create several config files and pick from them using *-c*. If you only wish to control the specific tests that are to be run (and not their parameters) then using *-s* or *-t* on the command line may be more appropriate.

1.1.1 Skipping Tests

The bandit config may contain optional lists of test IDs to either include (*tests*) or exclude (*skips*). These lists are equivalent to using *-t* and *-s* on the command line. If only *tests* is given then bandit will include only those tests, effectively excluding all other tests. If only *skips* is given then bandit will include all tests not in the skips list. If both

are given then bandit will include only tests in *tests* and then remove *skips* from that set. It is an error to include the same test ID in both *tests* and *skips*.

Note that command line options *-t*/*-s* can still be used in conjunction with *tests* and *skips* given in a config. The result is to concatenate *-t* with *tests* and likewise for *-s* and *skips* before working out the tests to run.

1.1.2 Generating a Config

Bandit ships the tool *bandit-config-generator* designed to take the leg work out of configuration. This tool can generate a configuration file automatically. The generated configuration will include default config blocks for all detected test and blacklist plugins. This data can then be deleted or edited as needed to produce a minimal config as desired. The config generator supports *-t* and *-s* command line options to specify a list of test IDs that should be included or excluded respectively. If no options are given then the generated config will not include *tests* or *skips* sections (but will provide a complete list of all test IDs for reference when editing).

1.1.3 Configuring Test Plugins

Bandit's configuration file is written in YAML and options for each plugin test are provided under a section named to match the test method. For example, given a test plugin called 'try_except_pass' its configuration section might look like the following:

```
try_except_pass:
  check_typed_exception: True
```

The specific content of the configuration block is determined by the plugin test itself. See the plugin test list for complete information on configuring each one.

1.2 Bandit Test Plugins

Bandit supports many different tests to detect various security issues in python code. These tests are created as plugins and new ones can be created to extend the functionality offered by bandit today.

1.2.1 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.
- Create a new Python source file to contain your test, you can reference existing tests for examples.
- 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')
- Register your plugin using the *bandit.plugins* entry point, see example.
- 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.

1.2.2 Config Generation

In Bandit 1.0+ config files are optional. Plugins that need config settings are required to implement a module global *gen_config* function. This function is called with a single parameter, the test plugin name. It should return a dictionary with keys being the config option names and values being the default settings for each option. An example *gen_config* might look like the following:

```
def gen_config(name):
    if name == 'try_except_continue':
        return {'check_typed_exception': False}
```

When no config file is specified, or when the chosen file has no section pertaining to a given plugin, *gen_config* will be called to provide defaults.

The config file generation tool *bandit-config-generator* will also call *gen_config* on all discovered plugins to produce template config blocks. If the defaults are acceptable then these blocks may be deleted to create a minimal configuration, or otherwise edited as needed. The above example would produce the following config snippet.

```
try_except_continue: {check_typed_exception: false}
```

1.2.3 Example Test Plugin

```
@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
```

1.2.4 Plugin ID Groupings

ID	Description
B1xx	misc tests
B2xx	application/framework misconfiguration
B3xx	blacklists (calls)
B4xx	blacklists (imports)
B5xx	cryptography
B6xx	injection
B7xx	XSS

1.2.5 Complete Test Plugin Listing

B604: any_other_function_with_shell_equals_true

B101: assert_used

B102: exec_used

B111: execute_with_run_as_root_equals_true

B201: flask_debug_true

B104: hardcoded_bind_all_interfaces

B106: hardcoded_password_funcarg

B107: hardcoded_password_default

B105: hardcoded_password_string

B608: hardcoded_sql_expressions

B108: hardcoded_tmp_directory

B701: jinja2_autoescape_false

B609: linux_commands_wildcard_injection

B601: paramiko_calls

B109: password_config_option_not_marked_secret

B501: request_with_no_cert_validation

B103: set_bad_file_permissions

B503: ssl_with_bad_defaults

B502: ssl_with_bad_version

B504: ssl_with_no_version

B605: start_process_with_a_shell

B606: start_process_with_no_shell

B607: start_process_with_partial_path

B602: subprocess_popen_with_shell_equals_true

B603: subprocess_without_shell_equals_true

B112: try_except_continue

B110: try_except_pass

B702: use_of_mako_templates

B505: weak_cryptographic_key

B506: yaml_load

1.3 Bandit Blacklist Plugins

Bandit supports built in functionality to implement blacklisting of imports and function calls, this functionality is provided by built in test 'B001'. This test may be filtered as per normal plugin filtering rules.

The exact calls and imports that are blacklisted, and the issues reported, are controlled by plugin methods with the entry point 'bandit.blacklists' and can be extended by third party plugins if desired. Blacklist plugins will be discovered by Bandit at startup and called. The returned results are combined into the final data set, subject to Bandit's normal test include/exclude rules allowing for fine grained control over blacklisted items. By convention, blacklisted calls should have IDs in the B3xx range and imports should have IDs in the B4xx range.

Plugin functions should return a dictionary mapping AST node types to lists of blacklist data. Currently the following node types are supported:

- Call, used for blacklisting calls.
- Import, used for blacklisting module imports (this also implicitly tests ImportFrom and Call nodes where the invoked function is Pythons built in '__import__()' method).

Items in the data lists are Python dictionaries with the following structure:

key	data meaning
'name'	The issue name string.
'id'	The bandit ID of the check, this must be unique and is used for filtering blacklist checks.
'qualnames'	A Python list of fully qualified name strings.
'message'	The issue message reported, this is a string that may contain the token '{name}' that will be substituted with the matched qualname in the final report.
'level'	The severity level reported.

A utility method bandit.blacklists.utils.build_conf_dict is provided to aid building these dictionaries.

Example

```
        >> Issue: [B317:blacklist] Using xml.sax.parse to parse untrusted XML data
        is known to be vulnerable to XML attacks. Replace xml.sax.parse with its
        defusedxml equivalent function.
            Severity: Medium   Confidence: High
            Location: ./examples/xml_sax.py:24
            23    sax.parseString(xmlString, ExampleContentHandler())
            24    sax.parse('notaxmlfilethatexists.xml', ExampleContentHandler)
            25
```

1.3.1 Complete Plugin Listing

blacklist_calls

blacklist_imports

New in version 0.17.0.

1.4 Bandit Report Formatters

Bandit supports many different formatters to output various security issues in python code. These formatters are created as plugins and new ones can be created to extend the functionality offered by bandit today.

1.4.1 Example Formatter

```
def report(manager, fileobj, sev_level, conf_level, lines=-1):
    result = bson.dumps(issues)
    with fileobj:
        fileobj.write(result)
```

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']}
```

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

```
    [entry_points]
    bandit.formatters =
        bson = bandit_bson:formatter
```

1.4.2 Complete Formatter Listing

csv

html

json

screen

text

xml

CHAPTER 2

Indices and tables

- genindex
- modindex
- search