Bandit Documentation

PyCQA

Feb 19, 2022

Contents

1 Getting Started 3

2 Indices and tables 51

Python Module Index 53

Index 55

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]
```

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 Suppressing Individual Lines

If you have lines in your code triggering vulnerability errors and you are certain that this is acceptable, they can be individually silenced by appending `# nosec` to the line:

```
# The following hash is not used in any security context. It is only used
# to generate unique values, collisions are acceptable and "data" is not
# coming from user-generated input
the_hash = md5(data).hexdigest()  # nosec
```

In such cases, it is good practice to add a comment explaining *why* a given line was excluded from security checks.

1.1.3 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).

1.1.4 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,
```

(continues on next page)

(continued from previous page)

```
            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

B101: assert_used

B101: Test for use of assert

This plugin test checks for the use of the Python `assert` keyword. It was discovered that some projects used assert to enforce interface constraints. However, assert is removed with compiling to optimised byte code (python -o producing *.pyo files). This caused various protections to be removed. The use of assert is also considered as general bad practice in OpenStack codebases.

Please see https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement for more info on `assert`

Config Options:

You can configure files that skip this check. This is often useful when you use assert statements in test cases.

```
assert_used:
  skips: ['*_test.py', 'test_*.py']
```

Example

```
>> Issue: Use of assert detected. The enclosed code will be removed when
   compiling to optimised byte code.
   Severity: Low   Confidence: High
   Location: ./examples/assert.py:1
1 assert logged_in
2 display_assets()
```

See also:

- https://bugs.launchpad.net/juniperopenstack/+bug/1456193
- https://bugs.launchpad.net/heat/+bug/1397883
- https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement

New in version 0.11.0.

B102: exec_used

B102: Test for the use of exec

This plugin test checks for the use of Python's *exec* method or keyword. The Python docs succinctly describe why the use of *exec* is risky.

Example

```
>> Issue: Use of exec detected.
   Severity: Medium   Confidence: High
   Location: ./examples/exec-py2.py:2
1 exec("do evil")
2 exec "do evil"
```

See also:

- https://docs.python.org/2/reference/simple_stmts.html#exec
- https://docs.python.org/3/library/functions.html#exec
- https://www.python.org/dev/peps/pep-0551/#background
- https://www.python.org/dev/peps/pep-0578/#suggested-audit-hook-locations

New in version 0.9.0.

B103: set_bad_file_permissions

B103: Test for setting permissive file permissions

POSIX based operating systems utilize a permissions model to protect access to parts of the file system. This model supports three roles "owner", "group" and "world" each role may have a combination of "read", "write" or "execute" flags sets. Python provides `chmod` to manipulate POSIX style permissions.

This plugin test looks for the use of `chmod` and will alert when it is used to set particularly permissive control flags. A MEDIUM warning is generated if a file is set to group executable and a HIGH warning is reported if a file is set world writable. Warnings are given with HIGH confidence.

Example

```
>> Issue: Probable insecure usage of temp file/directory.
   Severity: Medium   Confidence: Medium
   Location: ./examples/os-chmod-py2.py:15
14  os.chmod('/etc/hosts', 0o777)
15  os.chmod('/tmp/oh_hai', 0x1ff)
16  os.chmod('/etc/passwd', stat.S_IRWXU)

>> Issue: Chmod setting a permissive mask 0777 on file (key_file).
   Severity: High   Confidence: High
   Location: ./examples/os-chmod-py2.py:17
16  os.chmod('/etc/passwd', stat.S_IRWXU)
17  os.chmod(key_file, 0o777)
18
```

See also:

- https://security.openstack.org/guidelines/dg_apply-restrictive-file-permissions.html
- https://en.wikipedia.org/wiki/File_system_permissions
- https://security.openstack.org

New in version 0.9.0.

B104: hardcoded_bind_all_interfaces

B104: Test for binding to all interfaces

Binding to all network interfaces can potentially open up a service to traffic on unintended interfaces, that may not be properly documented or secured. This plugin test looks for a string pattern "0.0.0.0" that may indicate a hardcoded binding to all network interfaces.

Example

```
>> Issue: Possible binding to all interfaces.
   Severity: Medium   Confidence: Medium
   Location: ./examples/binding.py:4
3   s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4   s.bind(('0.0.0.0', 31137))
5   s.bind(('192.168.0.1', 8080))
```

See also:

- https://nvd.nist.gov/vuln/detail/CVE-2018-1281

New in version 0.9.0.

B105: hardcoded_password_string

`bandit.plugins.general_hardcoded_password.`**`hardcoded_password_string`**(*context*)

B105: Test for use of hard-coded password strings

The use of hard-coded passwords increases the possibility of password guessing tremendously. This plugin test looks for all string literals and checks the following conditions:

- assigned to a variable that looks like a password
- assigned to a dict key that looks like a password

- used in a comparison with a variable that looks like a password

Variables are considered to look like a password if they have match any one of:

- "password"
- "pass"
- "passwd"
- "pwd"
- "secret"
- "token"
- "secrete"

Note: this can be noisy and may generate false positives.

Config Options:

None

Example

```
>> Issue: Possible hardcoded password '(root)'
   Severity: Low   Confidence: Low
   Location: ./examples/hardcoded-passwords.py:5
4 def someFunction2(password):
5     if password == "root":
6         print("OK, logged in")
```

See also:

- https://www.owasp.org/index.php/Use_of_hard-coded_password

New in version 0.9.0.

B106: hardcoded_password_funcarg

`bandit.plugins.general_hardcoded_password.`**`hardcoded_password_funcarg`**(*context*)

B106: Test for use of hard-coded password function arguments

The use of hard-coded passwords increases the possibility of password guessing tremendously. This plugin test looks for all function calls being passed a keyword argument that is a string literal. It checks that the assigned local variable does not look like a password.

Variables are considered to look like a password if they have match any one of:

- "password"
- "pass"
- "passwd"
- "pwd"
- "secret"
- "token"
- "secrete"

Note: this can be noisy and may generate false positives.

Config Options:

None

Example

```
>> Issue: [B106:hardcoded_password_funcarg] Possible hardcoded
password: 'blerg'
   Severity: Low   Confidence: Medium
   Location: ./examples/hardcoded-passwords.py:16
15
16    doLogin(password="blerg")
```

See also:

- https://www.owasp.org/index.php/Use_of_hard-coded_password

New in version 0.9.0.

B107: hardcoded_password_default

`bandit.plugins.general_hardcoded_password.`**`hardcoded_password_default`**(*context*)

B107: Test for use of hard-coded password argument defaults

The use of hard-coded passwords increases the possibility of password guessing tremendously. This plugin test looks for all function definitions that specify a default string literal for some argument. It checks that the argument does not look like a password.

Variables are considered to look like a password if they have match any one of:

- "password"
- "pass"
- "passwd"
- "pwd"
- "secret"
- "token"
- "secrete"

Note: this can be noisy and may generate false positives.

Config Options:

None

Example

```
>> Issue: [B107:hardcoded_password_default] Possible hardcoded
password: 'Admin'
   Severity: Low   Confidence: Medium
   Location: ./examples/hardcoded-passwords.py:1

1    def someFunction(user, password="Admin"):
2      print("Hi " + user)
```

See also:

- https://www.owasp.org/index.php/Use_of_hard-coded_password

New in version 0.9.0.

B108: hardcoded_tmp_directory

B108: Test for insecure usage of tmp file/directory

Safely creating a temporary file or directory means following a number of rules (see the references for more details). This plugin test looks for strings starting with (configurable) commonly used temporary paths, for example:

- /tmp
- /var/tmp
- /dev/shm
- etc

Config Options:

This test plugin takes a similarly named config block, *hardcoded_tmp_directory*. The config block provides a Python list, *tmp_dirs*, that lists string fragments indicating possible temporary file paths. Any string starting with one of these fragments will report a MEDIUM confidence issue.

```
hardcoded_tmp_directory:
    tmp_dirs: ['/tmp', '/var/tmp', '/dev/shm']
```

Example

See also:

- https://security.openstack.org/guidelines/dg_using-temporary-files-securely.html

New in version 0.9.0.

B109: Test for a password based config option not marked secret

This plugin has been removed.

Passwords are sensitive and must be protected appropriately. In OpenStack Oslo there is an option to mark options "secret" which will ensure that they are not logged. This plugin detects usages of oslo configuration functions that appear to deal with strings ending in 'password' and flag usages where they have not been marked secret.

If such a value is found a MEDIUM severity error is generated. If 'False' or 'None' are explicitly set, Bandit will return a MEDIUM confidence issue. If Bandit can't determine the value of secret it will return a LOW confidence issue.

B110: try_except_pass

B110: Test for a pass in the except block

Errors in Python code bases are typically communicated using `Exceptions`. An exception object is 'raised' in the event of an error and can be 'caught' at a later point in the program, typically some error handling or logging action will then be performed.

However, it is possible to catch an exception and silently ignore it. This is illustrated with the following example

```
try:
  do_some_stuff()
except Exception:
  pass
```

This pattern is considered bad practice in general, but also represents a potential security issue. A larger than normal volume of errors from a service can indicate an attempt is being made to disrupt or interfere with it. Thus errors should, at the very least, be logged.

There are rare situations where it is desirable to suppress errors, but this is typically done with specific exception types, rather than the base Exception class (or no type). To accommodate this, the test may be configured to ignore 'try, except, pass' where the exception is typed. For example, the following would not generate a warning if the configuration option `checked_typed_exception` is set to False:

```
try:
  do_some_stuff()
except ZeroDivisionError:
  pass
```

Config Options:

```
try_except_pass:
  check_typed_exception: True
```

Example

```
>> Issue: Try, Except, Pass detected.
   Severity: Low   Confidence: High
   Location: ./examples/try_except_pass.py:4
3        a = 1
4    except:
5        pass
```

See also:

- https://security.openstack.org

New in version 0.13.0.

B111: Test for the use of rootwrap running as root

This plugin has been removed.

Running commands as root dramatically increase their potential risk. Running commands with restricted user privileges provides defense in depth against command injection attacks, or developer and configuration error. This plugin test checks for specific methods being called with a keyword parameter *run_as_root* set to True, a common OpenStack idiom.

B112: try_except_continue

B112: Test for a continue in the except block

Errors in Python code bases are typically communicated using `Exceptions`. An exception object is 'raised' in the event of an error and can be 'caught' at a later point in the program, typically some error handling or logging action will then be performed.

However, it is possible to catch an exception and silently ignore it while in a loop. This is illustrated with the following example

```
while keep_going:
  try:
    do_some_stuff()
  except Exception:
    continue
```

This pattern is considered bad practice in general, but also represents a potential security issue. A larger than normal volume of errors from a service can indicate an attempt is being made to disrupt or interfere with it. Thus errors should, at the very least, be logged.

There are rare situations where it is desirable to suppress errors, but this is typically done with specific exception types, rather than the base Exception class (or no type). To accommodate this, the test may be configured to ignore 'try, except, continue' where the exception is typed. For example, the following would not generate a warning if the configuration option `checked_typed_exception` is set to False:

```
while keep_going:
  try:
    do_some_stuff()
  except ZeroDivisionError:
    continue
```

Config Options:

```
try_except_continue:
  check_typed_exception: True
```

Example

```
>> Issue: Try, Except, Continue detected.
   Severity: Low   Confidence: High
   Location: ./examples/try_except_continue.py:5
4           a = i
5       except:
6           continue
```

See also:

- https://security.openstack.org

New in version 1.0.0.

B201: flask_debug_true

B201: Test for use of flask app with debug set to true

Running Flask applications in debug mode results in the Werkzeug debugger being enabled. This includes a feature that allows arbitrary code execution. Documentation for both Flask[1] and Werkzeug[2] strongly suggests that debug mode should never be enabled on production systems.

Operating a production server with debug mode enabled was the probable cause of the Patreon breach in 2015[3].

Example

[1] http://flask.pocoo.org/docs/1.0/quickstart/#debug-mode
[2] http://werkzeug.palletsprojects.com/en/0.15.x/debug/
[3] http://labs.detectify.com/post/130332638391/how-patreon-got-hacked-publicly-exposed-werkzeug

```
>> Issue: A Flask app appears to be run with debug=True, which exposes
the Werkzeug debugger and allows the execution of arbitrary code.
   Severity: High   Confidence: High
      Location: examples/flask_debug.py:10
      9 #bad
      10    app.run(debug=True)
      11
```

See also:

New in version 0.15.0.

B501: request_with_no_cert_validation

B501: Test for missing certificate validation

Encryption in general is typically critical to the security of many applications. Using TLS can greatly increase security by guaranteeing the identity of the party you are communicating with. This is accomplished by one or both parties presenting trusted certificates during the connection initialization phase of TLS.

When request methods are used certificates are validated automatically which is the desired behavior. If certificate validation is explicitly turned off Bandit will return a HIGH severity error.

Example

```
>> Issue: [request_with_no_cert_validation] Requests call with verify=False
disabling SSL certificate checks, security issue.
   Severity: High   Confidence: High
   Location: examples/requests-ssl-verify-disabled.py:4
3   requests.get('https://gmail.com', verify=True)
4   requests.get('https://gmail.com', verify=False)
5   requests.post('https://gmail.com', verify=True)
```

See also:

- https://security.openstack.org/guidelines/dg_move-data-securely.html
- https://security.openstack.org/guidelines/dg_validate-certificates.html

New in version 0.9.0.

B502: ssl_with_bad_version

`bandit.plugins.insecure_ssl_tls.`**`ssl_with_bad_version`**(*context*, *config*)

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.

It is worth noting that native support for TLS 1.2 is only available in more recent Python versions, specifically 2.7.9 and up, and 3.x

A note on 'SSLv23':

Amongst the available SSL/TLS versions provided by Python/pyOpenSSL there exists the option to use SSLv23. This very poorly named option actually means "use the highest version of SSL/TLS supported by both the server and client". This may (and should be) a version well in advance of SSL v2 or v3. Bandit can scan for the use of SSLv23 if desired, but its detection does not necessarily indicate a problem.

When using SSLv23 it is important to also provide flags to explicitly exclude bad versions of SSL/TLS from the protocol versions considered. Both the Python native and pyOpenSSL modules provide the `OP_NO_SSLv2` and `OP_NO_SSLv3` flags for this purpose.

Config Options:

```
ssl_with_bad_version:
    bad_protocol_versions:
        - PROTOCOL_SSLv2
        - SSLv2_METHOD
        - SSLv23_METHOD
        - PROTOCOL_SSLv3  # strict option
        - PROTOCOL_TLSv1  # strict option
        - SSLv3_METHOD    # strict option
        - TLSv1_METHOD    # strict option
```

Example

```
>> Issue: ssl.wrap_socket call with insecure SSL/TLS protocol version
identified, security issue.
   Severity: High   Confidence: High
   Location: ./examples/ssl-insecure-version.py:13
12  # strict tests
13  ssl.wrap_socket(ssl_version=ssl.PROTOCOL_SSLv3)
14  ssl.wrap_socket(ssl_version=ssl.PROTOCOL_TLSv1)
```

See also:

- `ssl_with_bad_defaults()`
- `ssl_with_no_version()`
- http://heartbleed.com/
- https://poodlebleed.com/
- https://security.openstack.org/
- https://security.openstack.org/guidelines/dg_move-data-securely.html

New in version 0.9.0.

B503: ssl_with_bad_defaults

`bandit.plugins.insecure_ssl_tls.ssl_with_bad_defaults(context, config)`

B503: Test for SSL use with bad defaults 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 Python methods with default parameter values that specify the use of broken SSL/TLS protocol versions. Currently, detection supports methods using Python's native SSL/TLS support and the pyOpenSSL module. A MEDIUM severity warning will be reported whenever known broken protocol versions are detected.

Config Options:

This test shares the configuration provided for the standard ../plugins/ssl_with_bad_version test, please refer to its documentation.

Example

```
>> Issue: Function definition identified with insecure SSL/TLS protocol
version by default, possible security issue.
   Severity: Medium   Confidence: Medium
   Location: ./examples/ssl-insecure-version.py:28
27
28  def open_ssl_socket(version=SSL.SSLv2_METHOD):
29      pass
```

See also:

- `ssl_with_bad_version()`
- `ssl_with_no_version()`
- http://heartbleed.com/
- https://poodlebleed.com/
- https://security.openstack.org/
- https://security.openstack.org/guidelines/dg_move-data-securely.html

New in version 0.9.0.

B504: ssl_with_no_version

`bandit.plugins.insecure_ssl_tls.ssl_with_no_version(context)`

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.

Config Options:

This test shares the configuration provided for the standard ../plugins/ssl_with_bad_version test, please refer to its documentation.

Example

```
>> Issue: ssl.wrap_socket call with no SSL/TLS protocol version
specified, the default SSLv23 could be insecure, possible security
issue.
   Severity: Low   Confidence: Medium
   Location: ./examples/ssl-insecure-version.py:23
22
23  ssl.wrap_socket()
24
```

See also:

- `ssl_with_bad_version()`
- `ssl_with_bad_defaults()`
- http://heartbleed.com/
- https://poodlebleed.com/
- https://security.openstack.org/
- https://security.openstack.org/guidelines/dg_move-data-securely.html

New in version 0.9.0.

B505: weak_cryptographic_key

B505: Test for weak cryptographic key use

As computational power increases, so does the ability to break ciphers with smaller key lengths. The recommended key length size for RSA and DSA algorithms is 2048 and higher. 1024 bits and below are now considered breakable. EC key length sizes are recommended to be 224 and higher with 160 and below considered breakable. This plugin test checks for use of any key less than those limits and returns a high severity error if lower than the lower threshold and a medium severity error for those lower than the higher threshold.

Example

```
>> Issue: DSA key sizes below 1024 bits are considered breakable.
   Severity: High   Confidence: High
   Location: examples/weak_cryptographic_key_sizes.py:36
35  # Also incorrect: without keyword args
36  dsa.generate_private_key(512,
37                           backends.default_backend())
38  rsa.generate_private_key(3,
```

See also:

- http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf
- https://security.openstack.org/guidelines/dg_strong-crypto.html

New in version 0.14.0.

B506: yaml_load

B506: Test for use of yaml load

This plugin test checks for the unsafe usage of the `yaml.load` function from the PyYAML package. The yaml.load function provides the ability to construct an arbitrary Python object, which may be dangerous if you receive a YAML document from an untrusted source. The function yaml.safe_load limits this ability to simple Python objects like integers or lists.

Please see http://pyyaml.org/wiki/PyYAMLDocumentation#LoadingYAML for more information on `yaml.load` and yaml.safe_load

Example

> **>> Issue: [yaml_load] Use of unsafe yaml load. Allows instantiation of** arbitrary objects. Consider yaml.safe_load(). Severity: Medium Confidence: High Location: examples/yaml_load.py:5
>
> 4 ystr = yaml.dump({'a' : 1, 'b' : 2, 'c' : 3}) 5 y = yaml.load(ystr) 6 yaml.dump(y)

See also:

- http://pyyaml.org/wiki/PyYAMLDocumentation#LoadingYAML

New in version 1.0.0.

B507: ssh_no_host_key_verification

B507: Test for missing host key validation

Encryption in general is typically critical to the security of many applications. Using SSH can greatly increase security by guaranteeing the identity of the party you are communicating with. This is accomplished by one or both parties presenting trusted host keys during the connection initialization phase of SSH.

When paramiko methods are used, host keys are verified by default. If host key verification is disabled, Bandit will return a HIGH severity error.

Example

```
>> Issue: [B507:ssh_no_host_key_verification] Paramiko call with policy set
to automatically trust the unknown host key.
Severity: High   Confidence: Medium
Location: examples/no_host_key_verification.py:4
3   ssh_client = client.SSHClient()
4   ssh_client.set_missing_host_key_policy(client.AutoAddPolicy)
5   ssh_client.set_missing_host_key_policy(client.WarningPolicy)
```

New in version 1.5.1.

B601: paramiko_calls

B601: Test for shell injection within Paramiko

Paramiko is a Python library designed to work with the SSH2 protocol for secure (encrypted and authenticated) connections to remote machines. It is intended to run commands on a remote host. These commands are run within a shell on the target and are thus vulnerable to various shell injection attacks. Bandit reports a MEDIUM issue when it detects the use of Paramiko's "exec_command" method advising the user to check inputs are correctly sanitized.

Example

```
>> Issue: Possible shell injection via Paramiko call, check inputs are
   properly sanitized.
   Severity: Medium   Confidence: Medium
   Location: ./examples/paramiko_injection.py:4
3    # this is not safe
4    paramiko.exec_command('something; really; unsafe')
5
```

See also:

- https://security.openstack.org
- https://github.com/paramiko/paramiko
- https://www.owasp.org/index.php/Command_Injection

New in version 0.12.0.

B602: subprocess_popen_with_shell_equals_true

`bandit.plugins.injection_shell.`**`subprocess_popen_with_shell_equals_true`**(*context*, *config*)

B602: Test for use of popen with shell equals true

Python possesses many mechanisms to invoke an external executable. However, doing so may present a security issue if appropriate care is not taken to sanitize any user provided or variable input.

This plugin test is part of a family of tests built to check for process spawning and warn appropriately. Specifically, this test looks for the spawning of a subprocess using a command shell. This type of subprocess invocation is dangerous as it is vulnerable to various shell injection attacks. Great care should be taken to sanitize all input in order to mitigate this risk. Calls of this type are identified by a parameter of 'shell=True' being given.

Additionally, this plugin scans the command string given and adjusts its reported severity based on how it is presented. If the command string is a simple static string containing no special shell characters, then the resulting issue has low severity. If the string is static, but contains shell formatting characters or wildcards, then the reported issue is medium. Finally, if the string is computed using Python's string manipulation or formatting operations, then the reported issue has high severity. These severity levels reflect the likelihood that the code is vulnerable to injection.

See also:

- ../plugins/linux_commands_wildcard_injection
- ../plugins/subprocess_without_shell_equals_true
- ../plugins/start_process_with_no_shell
- ../plugins/start_process_with_a_shell
- ../plugins/start_process_with_partial_path

Config Options:

This plugin test shares a configuration with others in the same family, namely *shell_injection*. This configuration is divided up into three sections, *subprocess*, *shell* and *no_shell*. They each list Python calls that spawn subprocesses, invoke commands within a shell, or invoke commands without a shell (by replacing the calling process) respectively.

This plugin specifically scans for methods listed in *subprocess* section that have shell=True specified.

```
shell_injection:

    # Start a process using the subprocess module, or one of its
    wrappers.
    subprocess:
        - subprocess.Popen
        - subprocess.call
```

Example

```
>> Issue: subprocess call with shell=True seems safe, but may be
changed in the future, consider rewriting without shell
   Severity: Low   Confidence: High
   Location: ./examples/subprocess_shell.py:21
20  subprocess.check_call(['/bin/ls', '-l'], shell=False)
21  subprocess.check_call('/bin/ls -l', shell=True)
22

>> Issue: call with shell=True contains special shell characters,
consider moving extra logic into Python code
   Severity: Medium   Confidence: High
   Location: ./examples/subprocess_shell.py:26
25
26  subprocess.Popen('/bin/ls *', shell=True)
27  subprocess.Popen('/bin/ls %s' % ('something',), shell=True)

>> Issue: subprocess call with shell=True identified, security issue.
   Severity: High   Confidence: High
   Location: ./examples/subprocess_shell.py:27
26  subprocess.Popen('/bin/ls *', shell=True)
27  subprocess.Popen('/bin/ls %s' % ('something',), shell=True)
28  subprocess.Popen('/bin/ls {}'.format('something'), shell=True)
```

See also:

- https://security.openstack.org
- https://docs.python.org/3/library/subprocess.html#frequently-used-arguments
- https://security.openstack.org/guidelines/dg_use-subprocess-securely.html
- https://security.openstack.org/guidelines/dg_avoid-shell-true.html

New in version 0.9.0.

B603: subprocess_without_shell_equals_true

`bandit.plugins.injection_shell.`**`subprocess_without_shell_equals_true`**(*context*, *config*)

B603: Test for use of subprocess without shell equals true

Python possesses many mechanisms to invoke an external executable. However, doing so may present a security issue if appropriate care is not taken to sanitize any user provided or variable input.

This plugin test is part of a family of tests built to check for process spawning and warn appropriately. Specifically, this test looks for the spawning of a subprocess without the use of a command shell. This type of

subprocess invocation is not vulnerable to shell injection attacks, but care should still be taken to ensure validity of input.

Because this is a lesser issue than that described in *subprocess_popen_with_shell_equals_true* a LOW severity warning is reported.

See also:

- ../plugins/linux_commands_wildcard_injection
- ../plugins/subprocess_popen_with_shell_equals_true
- ../plugins/start_process_with_no_shell
- ../plugins/start_process_with_a_shell
- ../plugins/start_process_with_partial_path

Config Options:

This plugin test shares a configuration with others in the same family, namely *shell_injection*. This configuration is divided up into three sections, *subprocess*, *shell* and *no_shell*. They each list Python calls that spawn subprocesses, invoke commands within a shell, or invoke commands without a shell (by replacing the calling process) respectively.

This plugin specifically scans for methods listed in *subprocess* section that have shell=False specified.

```
shell_injection:
    # Start a process using the subprocess module, or one of its
    wrappers.
    subprocess:
        - subprocess.Popen
        - subprocess.call
```

Example

```
>> Issue: subprocess call - check for execution of untrusted input.
   Severity: Low   Confidence: High
   Location: ./examples/subprocess_shell.py:23
22
23    subprocess.check_output(['/bin/ls', '-l'])
24
```

See also:

- https://security.openstack.org
- https://docs.python.org/3/library/subprocess.html#frequently-used-arguments
- https://security.openstack.org/guidelines/dg_avoid-shell-true.html
- https://security.openstack.org/guidelines/dg_use-subprocess-securely.html

New in version 0.9.0.

B604: any_other_function_with_shell_equals_true

`bandit.plugins.injection_shell.`**`any_other_function_with_shell_equals_true`**(*context*, *config*)

B604: Test for any function with shell equals true

Python possesses many mechanisms to invoke an external executable. However, doing so may present a security issue if appropriate care is not taken to sanitize any user provided or variable input.

This plugin test is part of a family of tests built to check for process spawning and warn appropriately. Specifically, this plugin test interrogates method calls for the presence of a keyword parameter *shell* equalling true. It is related to detection of shell injection issues and is intended to catch custom wrappers to vulnerable methods that may have been created.

See also:

- ../plugins/linux_commands_wildcard_injection
- ../plugins/subprocess_popen_with_shell_equals_true
- ../plugins/subprocess_without_shell_equals_true
- ../plugins/start_process_with_no_shell
- ../plugins/start_process_with_a_shell
- ../plugins/start_process_with_partial_path

Config Options:

This plugin test shares a configuration with others in the same family, namely *shell_injection*. This configuration is divided up into three sections, *subprocess*, *shell* and *no_shell*. They each list Python calls that spawn subprocesses, invoke commands within a shell, or invoke commands without a shell (by replacing the calling process) respectively.

Specifically, this plugin excludes those functions listed under the subprocess section, these methods are tested in a separate specific test plugin and this exclusion prevents duplicate issue reporting.

```
shell_injection:
    # Start a process using the subprocess module, or one of its
    wrappers.
    subprocess: [subprocess.Popen, subprocess.call,
                 subprocess.check_call, subprocess.check_output
                 execute_with_timeout]
```

Example

```
>> Issue: Function call with shell=True parameter identified, possible
security issue.
   Severity: Medium   Confidence: High
   Location: ./examples/subprocess_shell.py:9
8 pop('/bin/gcc --version', shell=True)
9 Popen('/bin/gcc --version', shell=True)
10
```

See also:

- https://security.openstack.org/guidelines/dg_avoid-shell-true.html
- https://security.openstack.org/guidelines/dg_use-subprocess-securely.html

New in version 0.9.0.

B605: start_process_with_a_shell

`bandit.plugins.injection_shell.start_process_with_a_shell(context, config)`

B605: Test for starting a process with a shell

Python possesses many mechanisms to invoke an external executable. However, doing so may present a security issue if appropriate care is not taken to sanitize any user provided or variable input.

This plugin test is part of a family of tests built to check for process spawning and warn appropriately. Specifically, this test looks for the spawning of a subprocess using a command shell. This type of subprocess invocation is dangerous as it is vulnerable to various shell injection attacks. Great care should be taken to sanitize all input in order to mitigate this risk. Calls of this type are identified by the use of certain commands which are known to use shells. Bandit will report a LOW severity warning.

See also:

- ../plugins/linux_commands_wildcard_injection
- ../plugins/subprocess_without_shell_equals_true
- ../plugins/start_process_with_no_shell
- ../plugins/start_process_with_partial_path
- ../plugins/subprocess_popen_with_shell_equals_true

Config Options:

This plugin test shares a configuration with others in the same family, namely *shell_injection*. This configuration is divided up into three sections, *subprocess*, *shell* and *no_shell*. They each list Python calls that spawn subprocesses, invoke commands within a shell, or invoke commands without a shell (by replacing the calling process) respectively.

This plugin specifically scans for methods listed in *shell* section.

```
shell_injection:
    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
```

Example

```
>> Issue: Starting a process with a shell: check for injection.
   Severity: Low   Confidence: Medium
   Location: examples/os_system.py:3
2
3   os.system('/bin/echo hi')
```

See also:

- https://security.openstack.org
- https://docs.python.org/3/library/os.html#os.system
- https://docs.python.org/3/library/subprocess.html#frequently-used-arguments
- https://security.openstack.org/guidelines/dg_use-subprocess-securely.html

New in version 0.10.0.

B606: start_process_with_no_shell

`bandit.plugins.injection_shell.`**`start_process_with_no_shell`**(*context*, *config*)

B606: Test for starting a process with no shell

Python possesses many mechanisms to invoke an external executable. However, doing so may present a security issue if appropriate care is not taken to sanitize any user provided or variable input.

This plugin test is part of a family of tests built to check for process spawning and warn appropriately. Specifically, this test looks for the spawning of a subprocess in a way that doesn't use a shell. Although this is generally safe, it maybe useful for penetration testing workflows to track where external system calls are used. As such a LOW severity message is generated.

See also:

- ../plugins/linux_commands_wildcard_injection
- ../plugins/subprocess_without_shell_equals_true
- ../plugins/start_process_with_a_shell
- ../plugins/start_process_with_partial_path
- ../plugins/subprocess_popen_with_shell_equals_true

Config Options:

This plugin test shares a configuration with others in the same family, namely *shell_injection*. This configuration is divided up into three sections, *subprocess*, *shell* and *no_shell*. They each list Python calls that spawn subprocesses, invoke commands within a shell, or invoke commands without a shell (by replacing the calling process) respectively.

This plugin specifically scans for methods listed in *no_shell* section.

```
shell_injection:
    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
```

(continues on next page)

(continued from previous page)

```
        - os.spawnve
        - os.spawnvp
        - os.spawnvpe
        - os.startfile
```

Example

```
>> Issue: [start_process_with_no_shell] Starting a process without a
   shell.
   Severity: Low   Confidence: Medium
   Location: examples/os-spawn.py:8
7   os.spawnv(mode, path, args)
8   os.spawnve(mode, path, args, env)
9   os.spawnvp(mode, file, args)
```

See also:

- https://security.openstack.org
- https://docs.python.org/3/library/os.html#os.system
- https://docs.python.org/3/library/subprocess.html#frequently-used-arguments
- https://security.openstack.org/guidelines/dg_use-subprocess-securely.html

New in version 0.10.0.

B607: start_process_with_partial_path

`bandit.plugins.injection_shell.`**`start_process_with_partial_path`**(*context*, *config*)

B607: Test for starting a process with a partial path

Python possesses many mechanisms to invoke an external executable. If the desired executable path is not fully qualified relative to the filesystem root then this may present a potential security risk.

In POSIX environments, the *PATH* environment variable is used to specify a set of standard locations that will be searched for the first matching named executable. While convenient, this behavior may allow a malicious actor to exert control over a system. If they are able to adjust the contents of the *PATH* variable, or manipulate the file system, then a bogus executable may be discovered in place of the desired one. This executable will be invoked with the user privileges of the Python process that spawned it, potentially a highly privileged user.

This test will scan the parameters of all configured Python methods, looking for paths that do not start at the filesystem root, that is, do not have a leading '/' character.

Config Options:

This plugin test shares a configuration with others in the same family, namely *shell_injection*. This configuration is divided up into three sections, *subprocess*, *shell* and *no_shell*. They each list Python calls that spawn subprocesses, invoke commands within a shell, or invoke commands without a shell (by replacing the calling process) respectively.

This test will scan parameters of all methods in all sections. Note that methods are fully qualified and de-aliased prior to checking.

```
shell_injection:
    # Start a process using the subprocess module, or one of its
    wrappers.
    subprocess:
        - subprocess.Popen
        - subprocess.call

    # Start a process with a function vulnerable to shell injection.
    shell:
        - os.system
        - os.popen
        - popen2.Popen3
        - popen2.Popen4
        - commands.getoutput
        - commands.getstatusoutput
    # Start a process with a function that is not vulnerable to shell
    injection.
    no_shell:
        - os.execl
        - os.execle
```

Example

```
>> Issue: Starting a process with a partial executable path
Severity: Low   Confidence: High
Location: ./examples/partial_path_process.py:3
2    from subprocess import Popen as pop
3    pop('gcc --version', shell=False)
```

See also:

- https://security.openstack.org
- https://docs.python.org/3/library/os.html#process-management

New in version 0.13.0.

B608: hardcoded_sql_expressions

B608: Test for SQL injection

An SQL injection attack consists of insertion or "injection" of a SQL query via the input data given to an application. It is a very common attack vector. This plugin test looks for strings that resemble SQL statements that are involved in some form of string building operation. For example:

- "SELECT %s FROM derp;" % var
- "SELECT thing FROM " + tab
- "SELECT " + val + " FROM " + tab + . . .
- "SELECT {} FROM derp;".format(var)
- f"SELECT foo FROM bar WHERE id = {product}"

Unless care is taken to sanitize and control the input data when building such SQL statement strings, an injection attack becomes possible. If strings of this nature are discovered, a LOW confidence issue is reported. In order to boost

result confidence, this plugin test will also check to see if the discovered string is in use with standard Python DBAPI calls *execute* or *executemany*. If so, a MEDIUM issue is reported. For example:

- cursor.execute("SELECT %s FROM derp;" % var)

  Example

```
>> Issue: Possible SQL injection vector through string-based query
construction.
   Severity: Medium   Confidence: Low
   Location: ./examples/sql_statements_without_sql_alchemy.py:4
3 query = "DELETE FROM foo WHERE id = '%s'" % identifier
4 query = "UPDATE foo SET value = 'b' WHERE id = '%s'" % identifier
5
```

See also:

- https://www.owasp.org/index.php/SQL_Injection
- https://security.openstack.org/guidelines/dg_parameterize-database-queries.html

New in version 0.9.0.

B609: linux_commands_wildcard_injection

B609: Test for use of wildcard injection

Python provides a number of methods that emulate the behavior of standard Linux command line utilities. Like their Linux counterparts, these commands may take a wildcard "*" character in place of a file system path. This is interpreted to mean "any and all files or folders" and can be used to build partially qualified paths, such as "/home/user/*".

The use of partially qualified paths may result in unintended consequences if an unexpected file or symlink is placed into the path location given. This becomes particularly dangerous when combined with commands used to manipulate file permissions or copy data off of a system.

This test plugin looks for usage of the following commands in conjunction with wild card parameters:

- 'chown'
- 'chmod'
- 'tar'
- 'rsync'

As well as any method configured in the shell or subprocess injection test configurations.

Config Options:

This plugin test shares a configuration with others in the same family, namely *shell_injection*. This configuration is divided up into three sections, *subprocess*, *shell* and *no_shell*. They each list Python calls that spawn subprocesses, invoke commands within a shell, or invoke commands without a shell (by replacing the calling process) respectively.

This test will scan parameters of all methods in all sections. Note that methods are fully qualified and de-aliased prior to checking.

```
shell_injection:
    # Start a process using the subprocess module, or one of its wrappers.
    subprocess:
        - subprocess.Popen
        - subprocess.call
```

(continues on next page)

(continued from previous page)

```
    # Start a process with a function vulnerable to shell injection.
    shell:
        - os.system
        - os.popen
        - popen2.Popen3
        - popen2.Popen4
        - commands.getoutput
        - commands.getstatusoutput
    # Start a process with a function that is not vulnerable to shell
    injection.
    no_shell:
        - os.execl
        - os.execle
```

Example

```
>> Issue: Possible wildcard injection in call: subprocess.Popen
   Severity: High   Confidence: Medium
   Location: ./examples/wildcard-injection.py:8
7    o.popen2('/bin/chmod *')
8    subp.Popen('/bin/chown *', shell=True)
9

>> Issue: subprocess call - check for execution of untrusted input.
   Severity: Low   Confidence: High
   Location: ./examples/wildcard-injection.py:11
10   # Not vulnerable to wildcard injection
11   subp.Popen('/bin/rsync *')
12   subp.Popen("/bin/chmod *")
```

See also:

- https://security.openstack.org
- https://en.wikipedia.org/wiki/Wildcard_character
- http://www.defensecode.com/public/DefenseCode_Unix_WildCards_Gone_Wild.txt

New in version 0.9.0.

B610: django_extra_used

`bandit.plugins.django_sql_injection.`**`django_extra_used`**`(`*context*`)`

B610: Potential SQL injection on extra function

See also:

- https://docs.djangoproject.com/en/dev/topics/security/#sql-injection-protection

New in version 1.5.0.

B611: django_rawsql_used

`bandit.plugins.django_sql_injection.`**`django_rawsql_used`**`(`*context*`)`

B611: Potential SQL injection on RawSQL function

See also:

- https://docs.djangoproject.com/en/dev/topics/security/#sql-injection-protection

New in version 1.5.0.

B701: jinja2_autoescape_false

B701: Test for not auto escaping in jinja2

Jinja2 is a Python HTML templating system. It is typically used to build web applications, though appears in other places well, notably the Ansible automation system. When configuring the Jinja2 environment, the option to use autoescaping on input can be specified. When autoescaping is enabled, Jinja2 will filter input strings to escape any HTML content submitted via template variables. Without escaping HTML input the application becomes vulnerable to Cross Site Scripting (XSS) attacks.

Unfortunately, autoescaping is False by default. Thus this plugin test will warn on omission of an autoescape setting, as well as an explicit setting of false. A HIGH severity warning is generated in either of these scenarios.

Example

```
>> Issue: Using jinja2 templates with autoescape=False is dangerous and can
lead to XSS. Use autoescape=True to mitigate XSS vulnerabilities.
   Severity: High   Confidence: High
   Location: ./examples/jinja2_templating.py:11
10  templateEnv = jinja2.Environment(autoescape=False,
    loader=templateLoader)
11  Environment(loader=templateLoader,
12              load=templateLoader,
13              autoescape=False)
14

>> Issue: By default, jinja2 sets autoescape to False. Consider using
autoescape=True or use the select_autoescape function to mitigate XSS
vulnerabilities.
   Severity: High   Confidence: High
   Location: ./examples/jinja2_templating.py:15
14
15  Environment(loader=templateLoader,
16              load=templateLoader)
17
18  Environment(autoescape=select_autoescape(['html', 'htm', 'xml']),
19              loader=templateLoader)
```

See also:

- OWASP XSS
- https://realpython.com/blog/python/primer-on-jinja-templating/
- http://jinja.pocoo.org/docs/dev/api/#autoescaping
- https://security.openstack.org
- https://security.openstack.org/guidelines/dg_cross-site-scripting-xss.html

New in version 0.10.0.

B702: use_of_mako_templates

B702: Test for use of mako templates

Mako is a Python templating system often used to build web applications. It is the default templating system used in Pylons and Pyramid. Unlike Jinja2 (an alternative templating system), Mako has no environment wide variable escaping mechanism. Because of this, all input variables must be carefully escaped before use to prevent possible vulnerabilities to Cross Site Scripting (XSS) attacks.

Example

```
>> Issue: Mako templates allow HTML/JS rendering by default and are
inherently open to XSS attacks. Ensure variables in all templates are
properly sanitized via the 'n', 'h' or 'x' flags (depending on context).
For example, to HTML escape the variable 'data' do ${ data |h }.
   Severity: Medium   Confidence: High
   Location: ./examples/mako_templating.py:10
9
10  mako.template.Template("hern")
11  template.Template("hern")
```

See also:

- http://www.makotemplates.org/
- OWASP XSS
- https://security.openstack.org
- https://security.openstack.org/guidelines/dg_cross-site-scripting-xss.html

New in version 0.10.0.

B703: django_mark_safe

`bandit.plugins.django_xss.`**`django_mark_safe`**`(`*context*`)`

B703: Potential XSS on mark_safe function

See also:

- https://docs.djangoproject.com/en/dev/topics/security/#cross-site-scripting-xss-protection
- https://docs.djangoproject.com/en/dev/ref/utils/#module-django.utils.safestring
- https://docs.djangoproject.com/en/dev/ref/utils/#django.utils.html.format_html

New in version 1.5.0.

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 various Python calls known to be dangerous

This blacklist data checks for a number of Python calls known to have possible security implications. The following blacklist tests are run against any function calls encoutered in the scanned code base, triggered by encoutering ast.Call nodes.

B301: pickle

Pickle and modules that wrap it can be unsafe when used to deserialize untrusted data, possible security issue.

ID	Name	Calls	Severity
B301	pickle	• pickle.loads • pickle.load • pickle.Unpickler • cPickle.loads • cPickle.load • cPickle.Unpickler • dill.loads • dill.load • dill.Unpickler • shelve.open • shelve.DbfilenameShelf	Medium

B302: marshal

Deserialization with the marshal module is possibly dangerous.

ID	Name	Calls	Severity
B302	marshal	• marshal.load • marshal.loads	Medium

B303: md5

Use of insecure MD2, MD4, MD5, or SHA1 hash function.

ID	Name	Calls	Severity
B303	md5	• hashlib.md5 • hashlib.sha1 • Crypto.Hash.MD2.new • Crypto.Hash.MD4.new • Crypto.Hash.MD5.new • Crypto.Hash.SHA.new • Cryptodome.Hash.MD2.new • Cryptodome.Hash.MD4.new • Cryptodome.Hash.MD5.new • Cryptodome.Hash.SHA.new • cryptography.hazmat.primitives.hashes.MD5 • cryptography.hazmat.primitives.hashes.SHA1	Medium

B304 - B305: ciphers and modes

Use of insecure cipher or cipher mode. Replace with a known secure cipher such as AES.

ID	Name	Calls	Severity
B304	ciphers	• Crypto.Cipher.ARC2.new • Crypto.Cipher.ARC4.new • Crypto.Cipher.Blowfish.new • Crypto.Cipher.DES.new • Crypto.Cipher.XOR.new • Cryptodome.Cipher.ARC2.new • Cryptodome.Cipher.ARC4.new • Cryptodome.Cipher.Blowfish.new • Cryptodome.Cipher.DES.new • Cryptodome.Cipher.XOR.new • cryptography.hazmat.primitives.ciphers.algorithms.ARC4 • cryptography.hazmat.primitives.ciphers.algorithms.Blowfish • cryptography.hazmat.primitives.ciphers.algorithms.IDEA	High
B305	cipher_modes	• cryptography.hazmat.primitives.ciphers.modes.ECB	Medium

B306: mktemp_q

Use of insecure and deprecated function (mktemp).

ID	Name	Calls	Severity
B306	mktemp_q	• tempfile.mktemp	Medium

B307: eval

Use of possibly insecure function - consider using safer ast.literal_eval.

ID	Name	Calls	Severity
B307	eval	• eval	Medium

B308: mark_safe

Use of mark_safe() may expose cross-site scripting vulnerabilities and should be reviewed.

ID	Name	Calls	Severity
B308	mark_safe	• django.utils.safestring.mark_safe	Medium

B309: httpsconnection

Use of HTTPSConnection on older versions of Python prior to 2.7.9 and 3.4.3 do not provide security, see https://wiki.openstack.org/wiki/OSSN/OSSN-0033

ID	Name	Calls	Severity
B309	httpsconnection	• httplib.HTTPSConnection • http.client.HTTPSConnection • six.moves.http_client.HTTPSConnection	Medium

B310: urllib_urlopen

Audit url open for permitted schemes. Allowing use of 'file:" or custom schemes is often unexpected.

ID	Name	Calls	Severity
B310	urllib_urlopen	• urllib.urlopen • urllib.request.urlopen • urllib.urlretrieve • urllib.request.urlretrieve • urllib.URLopener • urllib.request.URLopener • urllib.FancyURLopener • urllib.request.FancyURLopener • urllib2.urlopen • urllib2.Request • six.moves.urllib.request.urlopen • six.moves.urllib.request.urlretrieve • six.moves.urllib.request.URLopener • six.moves.urllib.request.FancyURLopener	Medium

B311: random

Standard pseudo-random generators are not suitable for security/cryptographic purposes.

ID	Name	Calls	Severity
B311	random	• random.random • random.randrange • random.randint • random.choice • random.uniform • random.triangular	Low

B312: telnetlib

Telnet-related functions are being called. Telnet is considered insecure. Use SSH or some other encrypted protocol.

ID	Name	Calls	Severity
B312	telnetlib	• telnetlib.*	High

B313 - B320: XML

Most of this is based off of Christian Heimes' work on defusedxml: https://pypi.org/project/defusedxml/#defusedxml-sax

Using various XLM methods to parse untrusted XML data is known to be vulnerable to XML attacks. Methods should be replaced with their defusedxml equivalents.

ID	Name	Calls	Severity
B313	xml_bad_cElementTree	• xml.etree.cElementTree.parse • xml.etree.cElementTree.iterparse • xml.etree.cElementTree.fromstring • xml.etree.cElementTree.XMLParser	Medium
B314	xml_bad_ElementTree	• xml.etree.ElementTree.parse • xml.etree.ElementTree.iterparse • xml.etree.ElementTree.fromstring • xml.etree.ElementTree.XMLParser	Medium
B315	xml_bad_expatreader	• xml.sax.expatreader.create_parser	Medium
B316	xml_bad_expatbuilder	• xml.dom.expatbuilder.parse • xml.dom.expatbuilder.parseString	Medium
B317	xml_bad_sax	• xml.sax.parse • xml.sax.parseString • xml.sax.make_parser	Medium
B318	xml_bad_minidom	• xml.dom.minidom.parse • xml.dom.minidom.parseString	Medium
B319	xml_bad_pulldom	• xml.dom.pulldom.parse • xml.dom.pulldom.parseString	Medium
B320	xml_bad_etree	• lxml.etree.parse • lxml.etree.fromstring • lxml.etree.RestrictedElement • lxml.etree.GlobalParserTLS • lxml.etree.getDefaultParser • lxml.etree.check_docinfo	Medium

B321: ftplib

FTP-related functions are being called. FTP is considered insecure. Use SSH/SFTP/SCP or some other encrypted protocol.

ID	Name	Calls	Severity
B321	ftplib	• ftplib.*	High

B322: input

The input method in Python 2 will read from standard input, evaluate and run the resulting string as python source code. This is similar, though in many ways worse, than using eval. On Python 2, use raw_input instead, input is safe in Python 3.

ID	Name	Calls	Severity
B322	input	• input	High

B323: unverified_context

By default, Python will create a secure, verified ssl context for use in such classes as HTTPSConnection. However, it still allows using an insecure context via the _create_unverified_context that reverts to the previous behavior that does not validate certificates or perform hostname checks.

ID	Name	Calls	Severity
B323	unverified_context	• ssl._create_unverified_context	Medium

B325: tempnam

Use of os.tempnam() and os.tmpnam() is vulnerable to symlink attacks. Consider using tmpfile() instead.

For further information: https://docs.python.org/2.7/library/os.html#os.tempnam https://docs.python.org/3/whatsnew/3.0.html?highlight=tempnam https://bugs.python.org/issue17880

ID	Name	Calls	Severity
B325	tempnam	• os.tempnam • os.tmpnam	Medium

blacklist_imports

Blacklist various Python imports known to be dangerous

This blacklist data checks for a number of Python modules known to have possible security implications. The following blacklist tests are run against any import statements or calls encountered in the scanned code base.

Note that the XML rules listed here are mostly based off of Christian Heimes' work on defusedxml: https://pypi.org/project/defusedxml/

B401: import_telnetlib

A telnet-related module is being imported. Telnet is considered insecure. Use SSH or some other encrypted protocol.

ID	Name	Imports	Severity
B401	import_telnetlib	• telnetlib	high

B402: import_ftplib

A FTP-related module is being imported. FTP is considered insecure. Use SSH/SFTP/SCP or some other encrypted protocol.

ID	Name	Imports	Severity
B402	import_ftplib	• ftplib	high

B403: import_pickle

Consider possible security implications associated with these modules.

ID	Name	Imports	Severity
B403	import_pickle	• pickle • cPickle • dill • shelve	low

B404: import_subprocess

Consider possible security implications associated with these modules.

ID	Name	Imports	Severity
B404	import_subprocess	• subprocess	low

B405: import_xml_etree

Using various methods to parse untrusted XML data is known to be vulnerable to XML attacks. Replace vulnerable imports with the equivalent defusedxml package, or make sure defusedxml.defuse_stdlib() is called.

ID	Name	Imports	Severity
B405	import_xml_etree	• xml.etree.cElementTree • xml.etree.ElementTree	low

B406: import_xml_sax

Using various methods to parse untrusted XML data is known to be vulnerable to XML attacks. Replace vulnerable imports with the equivalent defusedxml package, or make sure defusedxml.defuse_stdlib() is called.

ID	Name	Imports	Severity
B406	import_xml_sax	• xml.sax	low

B407: import_xml_expat

Using various methods to parse untrusted XML data is known to be vulnerable to XML attacks. Replace vulnerable imports with the equivalent defusedxml package, or make sure defusedxml.defuse_stdlib() is called.

ID	Name	Imports	Severity
B407	import_xml_expat	• xml.dom.expatbuilder	low

B408: import_xml_minidom

Using various methods to parse untrusted XML data is known to be vulnerable to XML attacks. Replace vulnerable imports with the equivalent defusedxml package, or make sure defusedxml.defuse_stdlib() is called.

ID	Name	Imports	Severity
B408	import_xml_minidom	• xml.dom.minidom	low

B409: import_xml_pulldom

Using various methods to parse untrusted XML data is known to be vulnerable to XML attacks. Replace vulnerable imports with the equivalent defusedxml package, or make sure defusedxml.defuse_stdlib() is called.

ID	Name	Imports	Severity
B409	import_xml_pulldom	• xml.dom.pulldom	low

B410: import_lxml

Using various methods to parse untrusted XML data is known to be vulnerable to XML attacks. Replace vulnerable imports with the equivalent defusedxml package.

ID	Name	Imports	Severity
B410	import_lxml	• lxml	low

B411: import_xmlrpclib

XMLRPC is particularly dangerous as it is also concerned with communicating data over a network. Use defused.xmlrpc.monkey_patch() function to monkey-patch xmlrpclib and mitigate remote XML attacks.

ID	Name	Imports	Severity
B411	import_xmlrpclib	• xmlrpclib	high

B412: import_httpoxy

httpoxy is a set of vulnerabilities that affect application code running in CGI, or CGI-like environments. The use of CGI for web applications should be avoided to prevent this class of attack. More details are available at https://httpoxy.org/.

ID	Name	Imports	Severity
B412	import_httpoxy	• wsgiref.handlers.CGIHandler • twisted.web.twcgi.CGIScript	high

B413: import_pycrypto

pycrypto library is known to have publicly disclosed buffer overflow vulnerability https://github.com/dlitz/pycrypto/issues/176. It is no longer actively maintained and has been deprecated in favor of pyca/cryptography library.

ID	Name	Imports	Severity
B413	import_pycrypto	• Crypto.Cipher • Crypto.Hash • Crypto.IO • Crypto.Protocol • Crypto.PublicKey • Crypto.Random • Crypto.Signature • Crypto.Util	high

B414: import_pycryptodome

This import blacklist has been removed. The information here has been left for historical purposes.

pycryptodome is a direct fork of pycrypto that has not fully addressed the issues inherent in PyCrypto. It seems to exist, mainly, as an API compatible continuation of pycrypto and should be deprecated in favor of pyca/cryptography which has more support among the Python community.

ID	Name	Imports	Severity
B414	import_pycryptodome	• Cryptodome.Cipher • Cryptodome.Hash • Cryptodome.IO • Cryptodome.Protocol • Cryptodome.PublicKey • Cryptodome.Random • Cryptodome.Signature • Cryptodome.Util	high

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

CSV Formatter

This formatter outputs the issues in a comma separated values format.

Example

```
filename,test_name,test_id,issue_severity,issue_confidence,issue_text,
line_number,line_range,more_info
examples/yaml_load.py,blacklist_calls,B301,MEDIUM,HIGH,"Use of unsafe yaml
load. Allows instantiation of arbitrary objects. Consider yaml.safe_load().
",5,[5],https://bandit.readthedocs.io/en/latest/
```

New in version 0.11.0.

Changed in version 1.5.0: New field *more_info* added to output

custom

Custom Formatter

This formatter outputs the issues in custom machine-readable format.

default template: `{abspath}:{line}: {test_id}[bandit]: {severity}: {msg}`

Example

```
/usr/lib/python3.6/site-packages/openlp/core/utils/__init__.py:405: B310[bandit]:␣
↪MEDIUM: Audit url open for permitted schemes. Allowing use of file:/ or custom␣
↪schemes is often unexpected.
```

New in version 1.5.0.

html

HTML formatter

This formatter outputs the issues as HTML.

Example

```
<!DOCTYPE html>
<html>
<head>

<meta charset="UTF-8">

<title>
    Bandit Report
</title>

<style>

html * {
    font-family: "Arial", sans-serif;
}

pre {
    font-family: "Monaco", monospace;
}

.bordered-box {
    border: 1px solid black;
    padding-top:.5em;
    padding-bottom:.5em;
    padding-left:1em;
}

.metrics-box {
    font-size: 1.1em;
    line-height: 130%;
}

.metrics-title {
    font-size: 1.5em;
    font-weight: 500;
    margin-bottom: .25em;
}

.issue-description {
    font-size: 1.3em;
    font-weight: 500;
}

.candidate-issues {
    margin-left: 2em;
    border-left: solid 1px; LightGray;
    padding-left: 5%;
    margin-top: .2em;
    margin-bottom: .2em;
}

.issue-block {
    border: 1px solid LightGray;
    padding-left: .5em;
    padding-top: .5em;
    padding-bottom: .5em;
```

(continues on next page)

(continued from previous page)

```
    margin-bottom: .5em;
}

.issue-sev-high {
    background-color: Pink;
}

.issue-sev-medium {
    background-color: NavajoWhite;
}

.issue-sev-low {
    background-color: LightCyan;
}

</style>
</head>

<body>

<div id="metrics">
    <div class="metrics-box bordered-box">
        <div class="metrics-title">
            Metrics:<br>
        </div>
        Total lines of code: <span id="loc">9</span><br>
        Total lines skipped (#nosec): <span id="nosec">0</span>
    </div>
</div>




<br>
<div id="results">

<div id="issue-0">
<div class="issue-block issue-sev-medium">
    <b>yaml_load: </b> Use of unsafe yaml load. Allows
    instantiation of arbitrary objects. Consider yaml.safe_load().<br>
    <b>Test ID:</b> B506<br>
    <b>Severity: </b>MEDIUM<br>
    <b>Confidence: </b>HIGH<br>
    <b>File: </b><a href="examples/yaml_load.py"
    target="_blank">examples/yaml_load.py</a> <br>
    <b>More info: </b><a href="https://bandit.readthedocs.io/en/latest/
    plugins/yaml_load.html" target="_blank">
    https://bandit.readthedocs.io/en/latest/plugins/yaml_load.html</a>
    <br>

<div class="code">
<pre>
5       ystr = yaml.dump({'a' : 1, 'b' : 2, 'c' : 3})
6       y = yaml.load(ystr)
7       yaml.dump(y)
</pre>
</div>
```

(continues on next page)

(continued from previous page)

```
</div>
</div>

</div>

</body>
</html>
```

New in version 0.14.0.

json

JSON formatter

This formatter outputs the issues in JSON.

Example

```
{
  "errors": [],
  "generated_at": "2015-12-16T22:27:34Z",
  "metrics": {
    "_totals": {
      "CONFIDENCE.HIGH": 1,
      "CONFIDENCE.LOW": 0,
      "CONFIDENCE.MEDIUM": 0,
      "CONFIDENCE.UNDEFINED": 0,
      "SEVERITY.HIGH": 0,
      "SEVERITY.LOW": 0,
      "SEVERITY.MEDIUM": 1,
      "SEVERITY.UNDEFINED": 0,
      "loc": 5,
      "nosec": 0
    },
    "examples/yaml_load.py": {
      "CONFIDENCE.HIGH": 1,
      "CONFIDENCE.LOW": 0,
      "CONFIDENCE.MEDIUM": 0,
      "CONFIDENCE.UNDEFINED": 0,
      "SEVERITY.HIGH": 0,
      "SEVERITY.LOW": 0,
      "SEVERITY.MEDIUM": 1,
      "SEVERITY.UNDEFINED": 0,
      "loc": 5,
      "nosec": 0
    }
  },
  "results": [
    {
      "code": "4     ystr = yaml.dump({'a' : 1, 'b' : 2, 'c' : 3})\n5
                    y = yaml.load(ystr)\n6     yaml.dump(y)\n",
      "filename": "examples/yaml_load.py",
      "issue_confidence": "HIGH",
```

(continues on next page)

(continued from previous page)

```
      "issue_severity": "MEDIUM",
      "issue_text": "Use of unsafe yaml load. Allows instantiation of
                     arbitrary objects. Consider yaml.safe_load().\n",
      "line_number": 5,
      "line_range": [
        5
      ],
      "more_info": "https://bandit.readthedocs.io/en/latest/",
      "test_name": "blacklist_calls",
      "test_id": "B301"
    }
  ]
}
```

New in version 0.10.0.

screen

Screen formatter

This formatter outputs the issues as color coded text to screen.

Example

```
>> Issue: [B506: yaml_load] Use of unsafe yaml load. Allows
   instantiation of arbitrary objects. Consider yaml.safe_load().

   Severity: Medium   Confidence: High
   Location: examples/yaml_load.py:5
   More Info: https://bandit.readthedocs.io/en/latest/
4       ystr = yaml.dump({'a' : 1, 'b' : 2, 'c' : 3})
5       y = yaml.load(ystr)
6       yaml.dump(y)
```

New in version 0.9.0.

text

Text Formatter

This formatter outputs the issues as plain text.

Example

```
>> Issue: [B301:blacklist_calls] Use of unsafe yaml load. Allows
   instantiation of arbitrary objects. Consider yaml.safe_load().

   Severity: Medium   Confidence: High
   Location: examples/yaml_load.py:5
   More Info: https://bandit.readthedocs.io/en/latest/
4       ystr = yaml.dump({'a' : 1, 'b' : 2, 'c' : 3})
5       y = yaml.load(ystr)
6       yaml.dump(y)
```

New in version 0.9.0.

xml

XML Formatter

This formatter outputs the issues as XML.

Example

```
<?xml version='1.0' encoding='utf-8'?>
<testsuite name="bandit" tests="1"><testcase
classname="examples/yaml_load.py" name="blacklist_calls"><error
message="Use of unsafe yaml load. Allows instantiation of arbitrary
objects. Consider yaml.safe_load().&#10;" type="MEDIUM"
more_info="https://bandit.readthedocs.io/en/latest/">Test ID: B301
Severity: MEDIUM Confidence: HIGH Use of unsafe yaml load. Allows
instantiation of arbitrary objects. Consider yaml.safe_load().

Location examples/yaml_load.py:5</error></testcase></testsuite>
```

New in version 0.12.0.

yaml

YAML Formatter

This formatter outputs the issues in a yaml format.

Example

```
errors: []
generated_at: '2017-03-09T22:29:30Z'
metrics:
  _totals:
    CONFIDENCE.HIGH: 1
    CONFIDENCE.LOW: 0
    CONFIDENCE.MEDIUM: 0
    CONFIDENCE.UNDEFINED: 0
    SEVERITY.HIGH: 0
    SEVERITY.LOW: 0
    SEVERITY.MEDIUM: 1
    SEVERITY.UNDEFINED: 0
    loc: 9
    nosec: 0
  examples/yaml_load.py:
    CONFIDENCE.HIGH: 1
    CONFIDENCE.LOW: 0
    CONFIDENCE.MEDIUM: 0
    CONFIDENCE.UNDEFINED: 0
    SEVERITY.HIGH: 0
    SEVERITY.LOW: 0
    SEVERITY.MEDIUM: 1
    SEVERITY.UNDEFINED: 0
    loc: 9
    nosec: 0
results:
- code: '5     ystr = yaml.dump({''a'' : 1, ''b'' : 2, ''c'' : 3})\n
```

(continues on next page)

(continued from previous page)

```
        6     y = yaml.load(ystr)\n7     yaml.dump(y)\n'
  filename: examples/yaml_load.py
  issue_confidence: HIGH
  issue_severity: MEDIUM
  issue_text: Use of unsafe yaml load. Allows instantiation of arbitrary
              objects.
    Consider yaml.safe_load().
  line_number: 6
  line_range:
  - 6
  more_info: https://bandit.readthedocs.io/en/latest/
  test_id: B506
  test_name: yaml_load
```

New in version 1.5.0.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

bandit.blacklists.calls, 31
bandit.blacklists.imports, 39

f

bandit.formatters.csv, 44
bandit.formatters.custom, 44
bandit.formatters.html, 44
bandit.formatters.json, 47
bandit.formatters.screen, 48
bandit.formatters.text, 48
bandit.formatters.xml, 49
bandit.formatters.yaml, 49

p

bandit.plugins.app_debug, 13
bandit.plugins.asserts, 6
bandit.plugins.crypto_request_no_cert_validation, 14
bandit.plugins.exec, 7
bandit.plugins.general_bad_file_permissions, 7
bandit.plugins.general_bind_all_interfaces, 8
bandit.plugins.general_hardcoded_tmp, 11
bandit.plugins.injection_paramiko, 18
bandit.plugins.injection_sql, 26
bandit.plugins.injection_wildcard, 27
bandit.plugins.jinja2_templates, 29
bandit.plugins.mako_templates, 30
bandit.plugins.ssh_no_host_key_verification, 18
bandit.plugins.try_except_continue, 12
bandit.plugins.try_except_pass, 11
bandit.plugins.weak_cryptographic_key, 17
bandit.plugins.yaml_load, 17

Index

B

bandit.blacklists.calls (*module*), 31
bandit.blacklists.imports (*module*), 39
bandit.formatters.csv (*module*), 44
bandit.formatters.custom (*module*), 44
bandit.formatters.html (*module*), 44
bandit.formatters.json (*module*), 47
bandit.formatters.screen (*module*), 48
bandit.formatters.text (*module*), 48
bandit.formatters.xml (*module*), 49
bandit.formatters.yaml (*module*), 49
bandit.plugins.app_debug (*module*), 13
bandit.plugins.asserts (*module*), 6
bandit.plugins.crypto_request_no_cert_validation (*module*), 14
bandit.plugins.exec (*module*), 7
bandit.plugins.general_bad_file_permissions (*module*), 7
bandit.plugins.general_bind_all_interfaces (*module*), 8
bandit.plugins.general_hardcoded_tmp (*module*), 11
bandit.plugins.injection_paramiko (*module*), 18
bandit.plugins.injection_sql (*module*), 26
bandit.plugins.injection_wildcard (*module*), 27
bandit.plugins.jinja2_templates (*module*), 29
bandit.plugins.mako_templates (*module*), 30
bandit.plugins.ssh_no_host_key_verification (*module*), 18
bandit.plugins.try_except_continue (*module*), 12
bandit.plugins.try_except_pass (*module*), 11
bandit.plugins.weak_cryptographic_key (*module*), 17
bandit.plugins.yaml_load (*module*), 17