Secrets
ABAP
Apex
C
C++
CloudFormation
COBOL
C#
CSS
Flex
Go
HTML
Java
JavaScript
Kotlin
Objective C
PHP
PL/I
PL/SQL
Python
RPG
Ruby
Scala
Swift
Terraform
Text
TypeScript
T-SQL
VB.NET
VB6
XML

Python static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your PYTHON code

All rules 216 | Vulnerability 29 | Bug 55 | Security Hotspot 31 | Code Smell 101

Tags

Search by name...

Values assigned to variables should match their type annotations
Code Smell

Function return types should be consistent with their type hint
Code Smell

Character classes in regular expressions should not contain the same character twice
Code Smell

Type checks shouldn't be confusing
Code Smell

Regular expressions should not be too complicated
Code Smell

Builtins should not be shadowed by local variables
Code Smell

Implicit string and byte concatenations should not be confusing
Code Smell

Identity comparisons should not be used with cached typed
Code Smell

Expressions creating sets should not have duplicate values
Code Smell

Expressions creating dictionaries should not have duplicate keys
Code Smell

Special method "__exit__" should not re-raise the provided exception
Code Smell

Disabling CSRF protections is security-sensitive

Analyze your code

Security Hotspot | Critical | cwe sans-top25 django owasp flask

A cross-site request forgery (CSRF) attack occurs when a trusted user of a web application can be forced, by an attacker, to perform sensitive actions that he didn't intend, such as updating his profile or sending a message, more generally anything that can change the state of the application.

The attacker can trick the user/victim to click on a link, corresponding to the privileged action, or to visit a malicious web site that embeds a hidden web request and as web browsers automatically include cookies, the actions can be authenticated and sensitive.

Ask Yourself Whether

- The web application uses cookies to authenticate users.
- There exist sensitive operations in the web application that can be performed when the user is authenticated.
- The state / resources of the web application can be modified by doing HTTP POST or HTTP DELETE requests for example.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Protection against CSRF attacks is strongly recommended:
  - to be activated by default for all unsafe HTTP methods.
  - implemented, for example, with an unguessable CSRF token
- Of course all sensitive operations should not be performed with safe HTTP methods like `GET` which are designed to be used only for information retrieval.

Sensitive Code Example

For a Django application, the code is sensitive when,

- `django.middleware.csrf.CsrfViewMiddleware` is not used in the Django settings:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddlewa
    'django.middleware.common.CommonMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddl
    'django.contrib.messages.middleware.MessageMiddlewa
    'django.middleware.clickjacking.XFrameOptionsMiddle
] # Sensitive: django.middleware.csrf.CsrfViewMiddlewar
```

- the CSRF protection is disabled on a view:

Unused scope-limited definitions should be removed

Code Smell

Functions and methods should not have identical implementations

Code Smell

Unused private nested classes should be removed

Code Smell

String formatting should be used correctly

Code Smell

```
@csrf_exempt # Sensitive
def example(request):
    return HttpResponse("default")
```

For a Flask application, the code is sensitive when,

- the `WTF_CSRF_ENABLED` setting is set to `false`:

```
app = Flask(__name__)
app.config['WTF_CSRF_ENABLED'] = False # Sensitive
```

- the application doesn't use the `CSRFProtect` module:

```
app = Flask(__name__) # Sensitive: CSRFProtect is missi

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

- the CSRF protection is disabled on a view:

```
app = Flask(__name__)
csrf = CSRFProtect()
csrf.init_app(app)

@app.route('/example/', methods=['POST'])
@csrf.exempt # Sensitive
def example():
    return 'example '
```

- the CSRF protection is disabled on a form:

```
class unprotectedForm(FlaskForm):
    class Meta:
        csrf = False # Sensitive

    name = TextField('name')
    submit = SubmitField('submit')
```

Compliant Solution

For a Django application,

- it is recommended to protect all the views with `django.middleware.csrf.CsrfViewMiddleware`:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddlewa
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware', # Comp
    'django.contrib.auth.middleware.AuthenticationMiddl
    'django.contrib.messages.middleware.MessageMiddlewa
    'django.middleware.clickjacking.XFrameOptionsMiddle
]
```

- and to not disable the CSRF protection on specific views:

```
def example(request): # Compliant
    return HttpResponse("default")
```

For a Flask application,

- the `CSRFProtect` module should be used (and not disabled further with `WTF_CSRF_ENABLED` set to `false`):

```
app = Flask(__name__)
csrf = CSRFProtect()
csrf.init_app(app) # Compliant
```

- and it is recommended to not disable the CSRF protection on specific views or forms:

```
@app.route('/example/', methods=['POST']) # Compliant
def example():
    return 'example '

class unprotectedForm(FlaskForm):
    class Meta:
        csrf = True # Compliant

    name = TextField('name')
    submit = SubmitField('submit')
```

See

- OWASP Top 10 2021 Category A1 - Broken Access Control
- MITRE, CWE-352 - Cross-Site Request Forgery (CSRF)
- OWASP Top 10 2017 Category A6 - Security Misconfiguration
- OWASP: Cross-Site Request Forgery
- SANS Top 25 - Insecure Interaction Between Components

Available In:

sonarcloud | sonarqube