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...

attacks

Vulnerability

The number and name of arguments passed to a function should match its parameters

Bug

The "open" builtin function should be called with a valid mode

Bug

Only defined names should be listed in "__all__"

Bug

Calls should not be made to non-callable values

Bug

Property getter, setter and deleter methods should have the expected number of parameters

Bug

Special methods should have an expected number of parameters

Bug

Instance and class methods should have at least one positional parameter

Bug

Boolean expressions of exceptions should not be used in "except" statements

Bug

Caught Exceptions must derive from BaseException

Bug

Item operations should be done on objects supporting them

Bug

Raised Exceptions must derive from

Database queries should not be vulnerable to injection attacks

Analyze your code

Vulnerability Blocker injection cwe owasp sans-top25 sql

User-provided data, such as URL parameters, should always be considered untrusted and tainted. Constructing SQL queries directly from tainted data enables attackers to inject specially crafted values that change the initial meaning of the query itself. Successful database query injection attacks can read, modify, or delete sensitive information from the database and sometimes even shut it down or execute arbitrary operating system commands.

Typically, the solution is to use prepared statements and to bind variables to SQL query parameters with dedicated methods like `params`, which ensures that user-provided data will be properly escaped. Another solution is to validate every parameter used to build the query. This can be achieved by transforming string values to primitive types or by validating them against a white list of accepted values.

This rule supports: sqlite3, mysql, pymysql, psycopg2, pgdb, Django ORM and Flask-SQLAlchemy.

Noncompliant Code Example

Flask application

```
from flask import request
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import text
from database.users import User

@app.route('hello')
def hello():
    id = request.args.get("id")
    stmt = text("SELECT * FROM users where id=%s" % id) # Qu
    query = SQLAlchemy().session.query(User).from_statement(
    user = query.one()
    return "Hello %s" % user.username
```

Django application

```
from django.http import HttpResponse
from django.db import connection

def hello(request):
    id = request.GET.get("id", "")
    cursor = connection.cursor()
    cursor.execute("SELECT username FROM auth_user WHERE id=
    row = cursor.fetchone()
    return HttpResponse("Hello %s" % row[0])
```

Compliant Solution

Flask application

BaseException

Bug

Operators should be used on compatible types

Bug

Function arguments should be passed only once

Bug

Iterable unpacking, "for-in" loops and "yield from" should use an Iterable object

Bug

```
from flask import request
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import text
from database.users import User

@app.route('hello')
def hello():
    id = request.args.get("id")
    stmt = text("SELECT * FROM users where id=:id")
    query = SQLAlchemy().session.query(User).from_statement(
    user = query.one()
    return "Hello %s" % user.username
```

Django application

```
from django.http import HttpResponse
from django.db import connection

def hello(request):
    id = request.GET.get("id", "")
    cursor = connection.cursor()
    cursor.execute("SELECT username FROM auth_user WHERE id=
    row = cursor.fetchone()
    return HttpResponse("Hello %s" % row[0])
```

See

See

- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-89 - Improper Neutralization of Special Elements used in an SQL Command
- MITRE, CWE-943 - Improper Neutralization of Special Elements in Data Query Logic
- OWASP SQL Injection Prevention Cheat Sheet
- SANS Top 25 - Insecure Interaction Between Components

Available In:

sonarcloud sonarqube Developer Edition