sonar RULES

Products

- 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...*

Functions should not have too many lines of code
Code Smell

Track uses of "NOSONAR" comments
Code Smell

Track comments matching a regular expression
Code Smell

Statements should be on separate lines
Code Smell

Functions should not contain too many return statements
Code Smell

Files should not have too many lines of code
Code Smell

Lines should not be too long
Code Smell

Methods and properties that don't access instance data should be static
Code Smell

New-style classes should be used
Code Smell

Parentheses should not be used after certain keywords
Code Smell

Track "TODO" and "FIXME" comments that do not contain a reference to a person
Code Smell

Module names should comply with a naming convention
Code Smell

Formatting SQL queries is security-sensitive

Analyze your code

Security Hotspot | Major | cwe owasp sans-top25 bad-practice sql

Formatted SQL queries can be difficult to maintain, debug and can increase the risk of SQL injection when concatenating untrusted values into the query. However, this rule doesn't detect SQL injections (unlike rule {rule:python:S3649}), the goal is only to highlight complex/formatted queries.

Ask Yourself Whether

- Some parts of the query come from untrusted values (like user inputs).
- The query is repeated/duplicated in other parts of the code.
- The application must support different types of relational databases.

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

Recommended Secure Coding Practices

- Use parameterized queries, prepared statements, or stored procedures and bind variables to SQL query parameters.
- Consider using ORM frameworks if there is a need to have an abstract layer to access data.

Sensitive Code Example

```
from django.db import models
from django.db import connection
from django.db import connections
from django.db.models.expressions import RawSQL

value = input()


class MyUser(models.Model):
    name = models.CharField(max_length=200)


def query_my_user(request, params, value):
    with connection.cursor() as cursor:
        cursor.execute("{0}".format(value))  # Sensitive

    # https://docs.djangoproject.com/en/2.1/ref/models/expre

    RawSQL("select col from %s where mycol = %s and othercol

    # https://docs.djangoproject.com/en/2.1/ref/models/query

    MyUser.objects.extra(
        select={
            'mycol':  "select col from sometable here mycol
           select_params=(someparam,),
        },
    )
```

Comments should not be located at the end of lines of code

Code Smell

Lines should not end with trailing whitespaces

Code Smell

Files should contain an empty newline at the end

Code Smell

Long suffix "L" should be upper case

Code Smell

Compliant Solution

```
cursor = connection.cursor(prepared=True)
sql_insert_query = """ select col from sometable here mycol

select_tuple = (1, value)

cursor.execute(sql_insert_query, select_tuple) # Compliant,
connection.commit()
```

See

- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- MITRE, CWE-89 - Improper Neutralization of Special Elements used in an SQL Command
- MITRE, CWE-564 - SQL Injection: Hibernate
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-943 - Improper Neutralization of Special Elements in Data Query Logic
- SANS Top 25 - Insecure Interaction Between Components
- Derived from FindSecBugs rules Potential SQL/JPQL Injection (JPA), Potential SQL/JDOQL Injection (JDO), Potential SQL/HQL Injection (Hibernate)

Available In:

sonarcloud sonarqube