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

PHP static code analysis

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

All rules (268)   Vulnerability (40)   Bug (51)   Security Hotspot (33)   Code Smell (144)

Tags   Search by name...

I/O function calls should not be vulnerable to path injection attacks
Vulnerability

LDAP queries should not be vulnerable to injection attacks
Vulnerability

OS commands should not be vulnerable to command injection attacks
Vulnerability

Class of caught exception should be defined
Bug

Caught Exceptions must derive from Throwable
Bug

Raised Exceptions must derive from Throwable
Bug

"$this" should not be used in a static context
Bug

Hard-coded credentials are security-sensitive
Security Hotspot

Test class names should end with "Test"
Code Smell

Tests should include assertions
Code Smell

TestCases should contain tests
Code Smell

Variable variables should not be used
Code Smell

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 `bindParam`, 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: Native Database Extensions, PDO, Symfony/Doctrine, Laravel/Eloquent.

Noncompliant Code Example

```
function authenticate() {
  if( isset( $_POST[ 'Connect' ] ) ) {
    $login = $_POST[ 'login' ];
    $pass = $_POST[ 'pass' ];

    $query = "SELECT * FROM users WHERE login = '" . $login

    // If the special value "foo' OR 1=1 --" is passed as ei
    // Indeed, if it is passed as a user, the query becomes:
    // SELECT * FROM users WHERE user = 'foo' OR 1=1 --' AND
    // As '--' is the comment till end of line syntax in SQL
    // SELECT * FROM users WHERE user = 'foo' OR 1=1
    // which is equivalent to:
    // SELECT * FROM users WHERE 1=1
    // which is equivalent to:
    // SELECT * FROM users

    $con = getDatabaseConnection();
    $result = mysqli_query($con, $query);

    $authenticated = false;
    if ( $row = mysqli_fetch_row( $result ) ) {
      $authenticated = true;
    }
    mysqli_free_result( $result );
    return $authenticated;
  }
}
```

Compliant Solution

A new session should be created during user authentication

🔓 Vulnerability

Cipher algorithms should be robust

🔓 Vulnerability

Encryption algorithms should be used with secure mode and padding scheme

🔓 Vulnerability

Server hostnames should be verified during SSL/TLS connections

```
function authenticate() {
  if( isset( $_POST[ 'Connect' ] ) ) {
    $login = $_POST[ 'login' ];
    $pass = $_POST[ 'pass' ];

    $query = "SELECT * FROM users WHERE login = :login AND p

    $stmt = $pdo->prepare($query);
    $stmt->bindParam(":login", $login);
    $stmt->bindParam(":pass", $pass);
    $stmt->execute();

    $authenticated = false;
    if ( $stmt->rowCount() == 1 ) {
      $authenticated = true;
    }

    return $authenticated;
  }
}
```

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