PHP static code analysis

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

Vulnerability

"exit(...)" and "die(...)" statements should not be used
Bug

Functions and variables should not be defined outside of classes
Code Smell

Track lack of copyright and license headers
Code Smell

Octal values should not be used
Code Smell

Switch cases should end with an unconditional "break" statement
Code Smell

Session-management cookies should not be persistent
Vulnerability

Cryptographic RSA algorithms should always incorporate OAEP (Optimal Asymmetric Encryption Padding)
Vulnerability

SHA-1 and Message-Digest hash algorithms should not be used in secure contexts
Vulnerability

Assertions should not be made at the end of blocks expecting an exception
Bug

Regular expressions should be syntactically valid
Bug

Only one method invocation is expected when testing exceptions
Bug

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:php: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

```
$id = $_GET['id'];
mysql_connect('localhost', $username, $password) or die('Cou
mysql_select_db('myDatabase') or die('Could not select datab

$result = mysql_query("SELECT * FROM myTable WHERE id = " .

while ($row = mysql_fetch_object($result)) {
    echo $row->name;
}
```

Compliant Solution

```
$id = $_GET['id'];
try {
    $conn = new PDO('mysql:host=localhost;dbname=myDatabase'
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCE

    $stmt = $conn->prepare('SELECT * FROM myTable WHERE id =
    $stmt->execute(array('id' => $id));

    while($row = $stmt->fetch(PDO::FETCH_OBJ)) {
        echo $row->name;
    }
} catch(PDOException $e) {
    echo 'ERROR: ' . $e->getMessage();
}
```

Reading the Standard Input is security-sensitive

Security Hotspot

Using command line arguments is security-sensitive

Security Hotspot

Using Sockets is security-sensitive

Security Hotspot

Encrypting data is security-sensitive

Security Hotspot

Exceptions

No issue will be raised if one of the functions is called with hard-coded string (no concatenation) and this string does not contain a "$" sign.

```
$result = mysql_query("SELECT * FROM myTable WHERE id = 42")
```

The current implementation does not follow variables. It will only detect SQL queries which are concatenated or contain a `$` sign directly in the function call.

```
$query = "SELECT * FROM myTable WHERE id = " . $id;
$result = mysql_query($query);  // No issue will be raised e
```

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