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...

"$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

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
Vulnerability

Server certificates should be verified during SSL/TLS connections
Vulnerability

LDAP connections should be authenticated

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

Analyze your code

Vulnerability | Blocker | injection cwe owasp sans-top25

User-provided data, such as URL parameters, POST data payloads, or cookies, should always be considered untrusted and tainted. Constructing file system paths directly from tainted data could enable an attacker to inject specially crafted values, such as `'../'`, that change the initial path and, when accessed, resolve to a path on the filesystem where the user should normally not have access.

A successful attack might give an attacker the ability to read, modify, or delete sensitive information from the file system and sometimes even execute arbitrary operating system commands. This is often referred to as a "path traversal" or "directory traversal" attack.

The mitigation strategy should be based on the whitelisting of allowed paths or characters.

Noncompliant Code Example

```
$userId = $_GET["userId"];
$fileUUID = $_GET["fileUUID"];

if ( $_SESSION["userId"] == $userId ) {
  unlink("/storage/" . $userId . "/" . $fileUUID); // N
}
```

Compliant Solution

```
$userId = (int) $_GET["userId"];
$fileUUID = (int) $_GET["fileUUID"];

if ( $_SESSION["userId"] == $userId ) {
  unlink("/storage/" . $userId . "/" . $fileUUID);
}
```

See

- OWASP Top 10 2021 Category A1 - Broken Access Control
- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- OWASP Top 10 2017 Category A5 - Broken Access Control
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-22 - Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
- MITRE, CWE-99 - Improper Control of Resource Identifiers ('Resource Injection')

Vulnerability

Cryptographic keys should be robust

Vulnerability

Weak SSL/TLS protocols should not be used

Vulnerability

Regular expressions should not be vulnerable to Denial of Service attacks

Vulnerability

Hashes should include an unpredictable salt

Vulnerability

- MITRE, CWE-641 - Improper Restriction of Names for Files and Other Resources
- SANS Top 25 - Risky Resource Management

Available In:

sonarcloud | sonarqube Developer Edition