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...

XML parsers should not be vulnerable to XXE attacks
Vulnerability

A secure password should be used when connecting to a database
Vulnerability

XPath expressions should not be vulnerable to injection attacks
Vulnerability

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

Deserialization should not be vulnerable to injection attacks

Analyze your code

Vulnerability | Blocker | injection cwe sans-top25 owasp

User-provided data such as URL parameters, POST data payloads or cookies should always be considered untrusted and tainted. Deserialization based on data supplied by the user could result in two types of attacks:

- Remote code execution attacks, where the structure of the serialized data is changed to modify the behavior of the object being unserialized.
- Parameter tampering attacks, where data is modified to escalate privileges or change for example quantity or price of products.

The best way to protect against deserialization attacks is probably to challenge the use of the deserialization mechanism in the application. They are cases were the use of deserialization mechanism was not justified and created breaches (CVE-2017-9785).

If the use of deserialization mechanisms is valid in your context, the problem could be mitigated in any of the following ways:

- Instead of using a native data interchange format, use a safe, standard format such as untyped JSON or structured data approaches such as Google Protocol Buffers.
- To ensure integrity is not compromised, add a digital signature (HMAC) to the serialized data that is validated before deserialization (this is only valid if the client doesn’t need to modify the serialized data)
- As a last resort, restrict deserialization to be possible only to specific, whitelisted classes.

Noncompliant Code Example

```
$data = $_GET["data"];
$object = unserialize($data);
// ...
```

Compliant Solution

```
$data = $_GET["data"];

list($hash, $data) = explode('|', $data, 2);
$hash_confirm = hash_hmac("sha256", $data, "secret-key"

// Confirm that the data integrity is not compromised
if ($hash === $hash_confirm) {
  $object = unserialize($data);
  // ...
}
```

See

- OWASP Top 10 2021 Category A8 - Software and Data Integrity Failures
- OWASP Top 10 2017 Category A8 - Insecure Deserialization
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-502 - Deserialization of Untrusted Data
- SANS Top 25 - Risky Resource Management

Available In:

sonarcloud | sonarqube Developer Edition

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