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...

be used to end lines

Code Smell

More than one property should not be declared per statement

Code Smell

The "var" keyword should not be used

Code Smell

"<?php" and "<?=" tags should be used

Code Smell

File names should comply with a naming convention

Code Smell

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

Code Smell

Local variable and function parameter names should comply with a naming convention

Code Smell

Field names should comply with a naming convention

Code Smell

Lines should not end with trailing whitespaces

Code Smell

Files should contain an empty newline at the end

Code Smell

Modifiers should be declared in the correct order

Code Smell

An open curly brace should be located at the beginning of a line

## Using regular expressions is security-sensitive

Analyze your code

Security Hotspot | Critical

Using regular expressions is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2017-16021
- CVE-2018-13863
- CVE-2018-8926

Evaluating regular expressions against input strings is potentially an extremely CPU-intensive task. Specially crafted regular expressions such as `/(a+)+s/` will take several seconds to evaluate the input string `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabs`. The problem is that with every additional `a` character added to the input, the time required to evaluate the regex doubles. However, the equivalent regular expression, `a+s` (without grouping) is efficiently evaluated in milliseconds and scales linearly with the input size.

Evaluating such regular expressions opens the door to Regular expression Denial of Service (ReDoS) attacks. In the context of a web application, attackers can force the web server to spend all of its resources evaluating regular expressions thereby making the service inaccessible to genuine users.

This rule flags any execution of a hardcoded regular expression which has at least 3 characters and contains at at least two instances of any of the following characters `*+{`.

Example: `(a+)*`

The following functions are detected as executing regular expressions:

- PCRE: Perl style regular expressions: preg_filter, preg_grep, preg_match_all, preg_match, preg_replace_callback, preg_replace, preg_split
- POSIX extended, which are deprecated: ereg_replace, ereg, eregi_replace, eregi, split, spliti.
- fnmatch
- Any of the multibyte string regular expressions: mb_eregi_replace, mb_ereg_match, mb_ereg_replace_callback, mb_ereg_replace, mb_ereg_search_init, mb_ereg_search_pos, mb_ereg_search_regs, mb_ereg_search, mb_ereg, mb_eregi_replace, mb_eregi

Note that `ereg*` functions have been removed in PHP 7 and **PHP 5 end of life date is the 1st of January 2019. Using PHP 5 is dangerous as there will be no security fix**.

This rule's goal is to guide security code reviews.

### Ask Yourself Whether

- the executed regular expression is sensitive and a user can provide a string which will be analyzed by this regular expression.
- your regular expression engine performance decrease with specially crafted inputs and regular expressions.

Code Smell

**An open curly brace should be located at the end of a line**

Code Smell

**Tabulation characters should not be used**

Code Smell

**Method and function names should comply with a naming convention**

Code Smell

**Creating cookies with broadly defined "domain" flags is security-sensitive**

Security Hotspot

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

#### Recommended Secure Coding Practices

Do not set the constant `pcre.backtrack_limit` to a high value as it will increase the resource consumption of PCRE functions.

Check the error codes of PCRE functions via `preg_last_error`.

Check whether your regular expression engine (the algorithm executing your regular expression) has any known vulnerabilities. Search for vulnerability reports mentioning the one engine you're are using. Do not run vulnerable regular expressions on user input.

Use if possible a library which is not vulnerable to Redos Attacks such as Google Re2.

Remember also that a ReDos attack is possible if a user-provided regular expression is executed. This rule won't detect this kind of injection.

Avoid executing a user input string as a regular expression or use at least `preg_quote` to escape regular expression characters.

#### Exceptions

An issue will be created for the functions `mb_ereg_search_pos`, `mb_ereg_search_regs` and `mb_ereg_search` if and only if at least the first argument, i.e. the $pattern, is provided.

The current implementation does not follow variables. It will only detect regular expressions hard-coded directly in the function call.

```
$pattern = "/(a+)+/";
$result = eregi($pattern, $input);  // No issue will be
```

Some corner-case regular expressions will not raise an issue even though they might be vulnerable. For example: `(a|aa)+`, `(a|a?)+`.

It is a good idea to test your regular expression if it has the same pattern on both side of a "|".

#### See

- OWASP Top 10 2017 Category A1 - Injection
- MITRE, CWE-624 - Executable Regular Expression Error
- OWASP Regular expression Denial of Service - ReDoS

#### Deprecated

This rule is deprecated; use {rule:phpsecurity:S2631} instead.

Available In:

sonarcloud | sonarqube