




 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



## Java static code analysis

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

All rules632

Vulnerability53

Bug154

Security Hotspot36


Code Smell389

Quick Fix42


Tags ▾

Search by name... 🔍


Regular expressions should not overflow the stack

 Bug


Tests method should not be annotated with competing annotations

 Bug


Assertions should not be used in production code

 Bug


DateTimeFormatters should not use mismatched year and week numbers

 Bug


Unicode Grapheme Clusters should be avoided inside regex character classes

 Bug


Case insensitive Unicode regular expressions should enable the "UNICODE\_CASE" flag

 Bug


Assertions should not compare an object to itself

 Bug


Regex alternatives should not be redundant

 Bug

Alternatives in regular expressions should be grouped when used with anchors

 Bug


AssertJ methods setting the assertion context should come before an assertion


 Bug


AssertJ configuration should be applied

### Using slow regular expressions is security-sensitive

Analyze your code

 Security Hotspot

 Critical



cwe owasp regex

Most of the regular expression engines use `backtracking` to try all possible execution paths of the regular expression when evaluating an input, in some cases it can cause performance issues, called `catastrophic backtracking` situations. In the worst case, the complexity of the regular expression is exponential in the size of the input, this means that a small carefully-crafted input (like 20 chars) can trigger `catastrophic backtracking` and cause a denial of service of the application. Super-linear regex complexity can lead to the same impact too with, in this case, a large carefully-crafted input (thousands chars).

This rule determines the runtime complexity of a regular expression and informs you of the complexity if it is not linear.

Note that, due to improvements to the matching algorithm, some cases of exponential runtime complexity have become impossible when run using JDK 9 or later. In such cases, an issue will only be reported if the project's target Java version is 8 or earlier.

#### Ask Yourself Whether

- The input is user-controlled.
- The input size is not restricted to a small number of characters.
- There is no timeout in place to limit the regex evaluation time.

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

#### Recommended Secure Coding Practices

To avoid `catastrophic backtracking` situations, make sure that none of the following conditions apply to your regular expression.

In all of the following cases, `catastrophic backtracking` can only happen if the problematic part of the regex is followed by a pattern that can fail, causing the backtracking to actually happen. Note that when performing a full match (e.g. using `String.matches`), the end of the regex counts as a pattern that can fail because it will only succeed when the end of the string is reached.

- If you have a non-possessive repetition `r*` or `r*?`, such that the regex `r` could produce different possible matches (of possibly different lengths) on the same input, the worst case matching time can be exponential. This can be the case if `r` contains optional parts, alternations or additional repetitions (but not if the repetition is written in such a way that there's only one way to match it).
  - When using JDK 9 or later an optimization applies when the repetition is greedy and the entire regex does not contain any back references. In that case the runtime will only be polynomial (in case of nested repetitions) or even linear (in case of alternations or optional parts).
- If you have multiple non-possessive repetitions that can match the same contents and are consecutive or are only separated by an optional separator or a separator that can be matched by both of the repetitions, the worst case matching time can be polynomial ( $O(n^c)$  where `c` is the number of problematic repetitions). For example `a*b*` is not a problem because `a*` and `b*` match different things and `a*_a*` is not a problem because the repetitions are separated by a `'_'` and can't match that `'_'`. However, `a*a*` and `.*_.*` have quadratic runtime.

https://rules.sonarsource.com/java/RSPEC-5852

1/3

- If you're performing a partial match (such as by using `Matcher.find`, `String.split`, `String.replaceAll` etc.) and the regex is not anchored to the beginning of the string, quadratic runtime is especially hard to avoid because whenever a match fails, the regex engine will try again starting at the next index. This means that any unbounded repetition (even a possessive one), if it's followed by a pattern that can fail, can cause quadratic runtime on some inputs. For example `str.split("\\s*,")` will run in quadratic time on strings that consist entirely of spaces (or at least contain large sequences of spaces, not followed by a comma).

In order to rewrite your regular expression without these patterns, consider the following strategies:

- If applicable, define a maximum number of expected repetitions using the bounded quantifiers, like `{1,5}` instead of `+` for instance.
- Refactor nested quantifiers to limit the number of way the inner group can be matched by the outer quantifier, for instance this nested quantifier situation `(ba+)+` doesn't cause performance issues, indeed, the inner group can be matched only if there exists exactly one `b` char per repetition of the group.
- Optimize regular expressions with possessive quantifiers and atomic grouping.
- Use negated character classes instead of `.` to exclude separators where applicable. For example the quadratic regex `.*_.*` can be made linear by changing it to `[^_]*_.*`

- Solve the problem without regular expressions
- Use an alternative non-backtracking regex implementations such as Google's [RE2](#) or [RE2/J](#).
- Use multiple passes. This could mean pre- and/or post-processing the string manually before/after applying the regular expression to it or using multiple regular expressions. One example of this would be to replace `str.split("\\s*\\.\\s*")` with `str.split(",")` and then trimming the spaces from the strings as a second step.
- When using `Matcher.find()`, it is often possible to make the regex infallible by making all the parts that could fail optional, which will prevent backtracking. Of course this means that you'll accept more strings than intended, but this can be handled by using capturing groups to check whether the optional parts were matched or not and then ignoring the match if they weren't. For example the regex `x*y` could be replaced with `x*(y)?` and then the call to `matcher.find()` could be replaced with `matcher.find() && matcher.group(1) != null`.

The first regex evaluation will never end in JDK <= 9 and the second regex evaluation will never end in any versions of the JDK:

## Compliant Solution

```
java.util.regex.Pattern.compile("(a)+").matcher(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaa!").matches(); // Compliant

java.util.regex.Pattern.compile("(h|h|ih((i|a|c|c|a|i|i|j|b|
"hhchchcihchchciicichhchchcihchchiihichiciiihhchci"+
```

```
"cchhcichcchiihchhccciiccichcchiihchchhchccciicccchccci
"chicihhccicccchihhchchichchcihiicichcihccccicccicciiii
"chhhhiihchihccccchhiiiiiiicicichicichccciichhhchihcii
"ichicccchhicchicihhccchicciihchchcihhiccccccccihchhihi
"ihhhhihihicichihiiiihhhhhhhhchhichiicihhiiiihchccccchic
```

See

- [OWASP Top 10 2017 Category A1](#) - Injection
- [MITRE, CWE-400](#) - Uncontrolled Resource Consumption
- [MITRE, CWE-1333](#) - Inefficient Regular Expression Complexity
- [owasp.org](#) - OWASP Regular expression Denial of Service - ReDoS
- [stackstatus.net](#) - Outage Postmortem - July 20, 2016
- [regular-expressions.info](#) - Runaway Regular Expressions: Catastrophic Backtracking
- [docs.microsoft.com](#) - Backtracking with Nested Optional Quantifiers

Available In:

