TypeScript static code analysis

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

Bug

Constructors should not be declared inside interfaces
Bug

Errors should not be created without being thrown
Bug

Collection sizes and array length comparisons should make sense
Bug

All branches in a conditional structure should not have exactly the same implementation
Bug

Destructuring patterns should not be empty
Bug

The output of functions that don't return anything should not be used
Bug

Comma and logical OR operators should not be used in switch cases
Bug

Generators should "yield" something
Bug

"new" operators should be used with functions
Bug

Non-existent operators '=+', '=-' and '=!' should not be used
Bug

"NaN" should not be used in comparisons
Bug

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 if it is not linear.

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.

- If you have a 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).
- If you have multiple 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.
- If 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, 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.

A "for" loop update clause should move the counter in the right direction

Bug

Return values from functions without side effects should not be ignored

Bug

Special identifiers should not be bound or assigned

Bug

Values should not be uselessly incremented

Bug

- 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 by emulating *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 `[^_]*_.*`

Sometimes it's not possible to rewrite the regex to be linear while still matching what you want it to match. Especially when the regex is not anchored to the beginning of the string, for which it is quite hard to avoid quadratic runtimes. In those cases consider the following approaches:

- Solve the problem without regular expressions
- Use an alternative non-backtracking regex implementations such as Google's RE2 or node-re2.
- 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.
- 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 `str.match(regex)` could be replaced with `matched = str.match(regex)` and `matched[1] !== undefined`.

Sensitive Code Example

The regex evaluation will never end:

```
/(a+)+$/.test(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaa!"
); // Sensitive
```

Compliant Solution

Possessive quantifiers do not keep backtracking positions, thus can be used, if possible, to avoid performance issues. Unfortunately, they are not supported in JavaScript, but one can still mimick them using lookahead assertions and backreferences:

```
/((?=(a+))\2)+$/.test(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaa!"
); // Compliant
```

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:

sonarcloud sonarqube