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

C# static code analysis

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

All rules 409 | Vulnerability 34 | Bug 76 | Security Hotspot 28 | Code Smell 271 | Quick Fix 52

Tags

Search by name...

"protected" members
Code Smell

Underscores should be used to make large numbers readable
Code Smell

"ToString()" calls should not be redundant
Code Smell

"==" should not be used when "Equals" is overridden
Code Smell

An abstract class should have both abstract and concrete methods
Code Smell

Multiple variables should not be declared on the same line
Code Smell

Culture should be specified for "string" operations
Code Smell

"switch" statements should have at least 3 "case" clauses
Code Smell

break statements should not be used except for switch cases
Code Smell

String literals should not be duplicated
Code Smell

Files should contain an empty newline at the end
Code Smell

Unused "using" should be removed
Code Smell

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

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 `aaaaaaaaaaaaaaaaaaaaaaaaaaaaabs`. 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 at least two instances of any of the following characters: `*+{`.

Example: `(a+)*`

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.

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

Recommended Secure Coding Practices

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.

If the regular expression is vulnerable to ReDos attacks, mitigate the risk by using a "match timeout" to limit the time spent running the regular expression.

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.

Sensitive Code Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.Text.RegularExpressions;
using System.Web;
```

A close curly brace should be located at the beginning of a line

Code Smell

Tabulation characters should not be used

Code Smell

Methods and properties should be named in PascalCase

Code Smell

Track uses of in-source issue suppressions

Code Smell

```
namespace N
{
    public class RegularExpression
    {
        void Foo(RegexOptions options, TimeSpan matchTimeout
                 string replacement, MatchEvaluator evaluato
        {
            // All the following instantiations are Sensitiv
            new System.Text.RegularExpressions.Regex("(a+)+"
            new System.Text.RegularExpressions.Regex("(a+)+"
            new System.Text.RegularExpressions.Regex("(a+)+"

            // All the following static methods are Sensitiv
            System.Text.RegularExpressions.Regex.IsMatch(inp
            System.Text.RegularExpressions.Regex.IsMatch(inp
            System.Text.RegularExpressions.Regex.IsMatch(inp

            System.Text.RegularExpressions.Regex.Match(input
            System.Text.RegularExpressions.Regex.Match(input
            System.Text.RegularExpressions.Regex.Match(input

            System.Text.RegularExpressions.Regex.Matches(inp
            System.Text.RegularExpressions.Regex.Matches(inp
            System.Text.RegularExpressions.Regex.Matches(inp

            System.Text.RegularExpressions.Regex.Replace(inp
            System.Text.RegularExpressions.Regex.Replace(inp
            System.Text.RegularExpressions.Regex.Replace(inp
            System.Text.RegularExpressions.Regex.Replace(inp
            System.Text.RegularExpressions.Regex.Replace(inp
            System.Text.RegularExpressions.Regex.Replace(inp

            System.Text.RegularExpressions.Regex.Split(input
            System.Text.RegularExpressions.Regex.Split(input
            System.Text.RegularExpressions.Regex.Split(input
        }
    }
}
```

Exceptions

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:roslyn.sonaranalyzer.security.cs:S2631} instead.

Available In:

sonarcloud | sonarqube