- 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#

C# static code analysis

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

members
Code Smell

"Explicit" conversions of "foreach" loops should not be used
Code Smell

Instance members should not write to "static" fields
Code Smell

"IndexOf" checks should not be for positive numbers
Code Smell

Whitespace and control characters in string literals should be explicit
Code Smell

Properties should not make collection or array copies
Code Smell

Flags enumerations zero-value members should be named "None"
Code Smell

Overflow checking should not be disabled for "Enumerable.Sum"
Code Smell

Field-like events should not be virtual
Code Smell

Non-constant static fields should not be visible
Code Smell

Inappropriate casts should not be made
Code Smell

Constructors should only call non-overridable methods
Code Smell

Regular expressions should not be vulnerable to Denial of Service attacks

Analyze your code

Vulnerability | Critical | injection cwe owasp denial-of-service

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).

It is not recommended to construct a regular expression pattern from a user-controlled input, if no other choice, sanitize the input to remove/annihilate regex metacharacters.

Noncompliant Code Example

```
using System.Text.RegularExpressions;
using Microsoft.AspNetCore.Mvc;

namespace WebApplicationDotNetCore.Controllers
{
    public class RSPEC2631RegExpInjectionNoncompliantControl
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Validate(string regex, string i
        {
            bool match = Regex.IsMatch(input, regex); // Non

            return Content("Valid? " + match);
        }
    }
}
```

Compliant Solution

```
using System;
using System.Text.RegularExpressions;
using Microsoft.AspNetCore.Mvc;

namespace WebApplicationDotNetCore.Controllers
{
    public class RSPEC2631RegExpInjectionCompliantController
    {
        public IActionResult Index()
        {
```

"GC.Collect" should not be called

Code Smell

Methods should not be empty

Code Smell

Exceptions should not be thrown in finally blocks

Code Smell

Method overrides should not change parameter defaults

Code Smell

```
            return View();
        }

        public IActionResult Validate(string regex, string i
        {
            bool match = Regex.IsMatch(input, Regex.Escape(r

            return Content("Valid? " + match);
        }
    }
}
```

See

- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-400 - Uncontrolled Resource Consumption
- MITRE, CWE-1333 - Inefficient Regular Expression Complexity
- OWASP Regular expression Denial of Service - ReDoS

Available In:

sonarcloud | sonarqube Developer Edition