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...

Bug

Static fields should appear in the order they must be initialized
Bug

Classes directly extending "object" should not call "base" in "GetHashCode" or "Equals"
Bug

Anonymous delegates should not be used to unsubscribe from Events
Bug

Delegates should not be subtracted
Bug

"async" methods should not return "void"
Bug

"ThreadStatic" should not be used on non-static fields
Bug

"IDisposables" created in a "using" statement should not be returned
Bug

"ThreadStatic" fields should not be initialized
Bug

"Object.ReferenceEquals" should not be used for value types
Bug

Doubled prefix operators "!!" and "~~" should not be used
Bug

"=+" should not be used instead of "+="
Bug

"ValueTask" should be consumed correctly

Analyze your code

Code Smell | Critical | async-await

`ValueTask<TResult>` was introduced in .NET Core 2.0 to optimize memory allocation when functions return their results synchronously.

`ValueTask` and `ValueTask<TResult>` should **never** be used in the following ways as it could result in a race condition:

- Calling `await` multiple times on a `ValueTask / ValueTask<TResult>`*. The wrapped object may have been reused by another operation. This differs from `Task / Task<TResult>`, on which you can await multiple times and always get the same result.
- Calling `await` concurrently on a `ValueTask / ValueTask<TResult>`*. The underlying object is not thread safe. What's more, it has the same effect as awaiting multiple times a `ValueTask / ValueTask<TResult>`. This again differs from `Task / Task<TResult>`, which support concurrent `await`.
- Using `.Result` or `.GetAwaiter().GetResult()` without checking if the operation completed*. `IValueTaskSource / IValueTaskSource<TResult>` implementations are not required to block until the operation completes. On the other hand, `Task / Task<TResult>` blocks the call until the task completes.

It is recommended to use `ValueTask / ValueTask<TResult>` either by calling "await" on the function returning it, optionally calling `ConfigureAwait(false)` on it, or by calling `.AsTask()` on it.

This rule raises an issue when the following operations are performed on a `ValueTask / ValueTask<TResult>` instance:

- Awaiting the instance multiple times.
- Calling `AsTask` multiple times.
- Using `.Result` or `.GetAwaiter().GetResult()` multiple times
- Using `.Result` or `.GetAwaiter().GetResult()` when the operation has not yet completed
- Using more than one of these ways to consume the instance.

Noncompliant Code Example

```
ValueTask<int> vt = SomeValueTaskReturningMethodAsync();
int result = await vt;
int result2 = await vt; // Noncompliant, variable is awaited

int value = SomeValueTaskReturningMethodAsync().GetAwaiter()
```

Compliant Solution

```
int result = await SomeValueTaskReturningMethodAsync();

int result = await SomeValueTaskReturningMethodAsync().Confi

Task<int> t = SomeValueTaskReturningMethodAsync().AsTask();
```

"NaN" should not be used in comparisons

Bug

Conditionally executed code should be reachable

Bug

Null pointers should not be dereferenced

Bug

For-loop conditions should be true at least once

Bug

Exceptions

This rule does not raise any issue when a `ValueTask / ValueTask<TResult>` is awaited multiple time in a loop.

See

- ValueTask<TResult> official documentation
- Understanding the Whys, Whats, and Whens of ValueTask

Available In:

sonarlint | sonarcloud | sonarqube