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...

Code Smell

"async" and "await" should not be used as identifiers
Code Smell

TestCases should contain tests
Code Smell

Short-circuit logic should be used in boolean contexts
Code Smell

JWT should be signed and verified with strong cipher algorithms
Vulnerability

Cipher algorithms should be robust
Vulnerability

Encryption algorithms should be used with secure mode and padding scheme
Vulnerability

Insecure temporary file creation methods should not be used
Vulnerability

Server certificates should be verified during SSL/TLS connections
Vulnerability

LDAP connections should be authenticated
Vulnerability

Cryptographic keys should be robust
Vulnerability

Weak SSL/TLS protocols should not be used
Vulnerability

Cipher Block Chaining IVs should be

"IDisposables" should be disposed

Analyze your code

Bug | Blocker | cwe denial-of-service

When writing managed code, you don't need to worry about allocating or freeing memory: The garbage collector takes care of it. For efficiency reasons, some objects such as `Bitmap` use unmanaged memory, enabling for example the use of pointer arithmetic. Such objects have potentially huge unmanaged memory footprints, but will have tiny managed ones. Unfortunately, the garbage collector only sees the tiny managed footprint, and fails to reclaim the unmanaged memory (by calling `Bitmap`'s finalizer method) in a timely fashion.

Moreover, memory is not the only system resource which needs to be managed in a timely fashion: The operating system can only handle having so many file descriptors (e.g. `FileStream`) or sockets (e.g. `WebClient`) open at any given time. Therefore, it is important to `Dispose` of them as soon as they are no longer needed, rather than relying on the garbage collector to call these objects' finalizers at some nondeterministic point in the future.

This rule tracks `private` fields and local variables of the following `IDisposable` types, which are never disposed, closed, aliased, returned, or passed to other methods.

- `System.IO` namespace
  - `System.IO.FileStream`
  - `System.IO.StreamReader`
  - `System.IO.StreamWriter`
- `System.Net` namespace
  - `System.Net.WebClient`
- `System.Net.Sockets` namespace
  - `System.Net.Sockets.Socket`
  - `System.Net.Sockets.TcpClient`
  - `System.Net.Sockets.UdpClient`
- `System.Drawing` namespace
  - `System.Drawing.Image`
  - `System.Drawing.Bitmap`

which are either instantiated directly using the `new` operator, or using one of the following factory methods:

- `System.IO.File.Create()`
- `System.IO.File.Open()`
- `System.Drawing.Image.FromFile()`
- `System.Drawing.Image.FromStream()`

on both private fields and local variables.

Noncompliant Code Example

```
public class ResourceHolder
{
  private FileStream fs; // Noncompliant; Dispose or Close a

  public void OpenResource(string path)
  {
    this.fs = new FileStream(path, FileMode.Open);
  }
```

unpredictable

Vulnerability

Regular expressions should not be vulnerable to Denial of Service attacks

Vulnerability

Hashes should include an unpredictable salt

Vulnerability

Non-async "Task/Task<T>" methods should not return null

Bug

Calls to delegate's method

```
  public void WriteToFile(string path, string text)
  {
    var fs = new FileStream(path, FileMode.Open); // Noncomp
    var bytes = Encoding.UTF8.GetBytes(text);
    fs.Write(bytes, 0, bytes.Length);
  }
}
```

Compliant Solution

```
public class ResourceHolder : IDisposable
{
  private FileStream fs;

  public void OpenResource(string path)
  {
    this.fs = new FileStream(path, FileMode.Open);
  }

  public void Dispose()
  {
    this.fs.Dispose();
  }

  public void WriteToFile(string path, string text)
  {
    using (var fs = new FileStream(path, FileMode.Open))
    {
      var bytes = Encoding.UTF8.GetBytes(text);
      fs.Write(bytes, 0, bytes.Length);
    }
  }
}
```

Exceptions

`IDisposable` variables returned from a method or passed to other methods are ignored, as are local `IDisposables` that are initialized with other `IDisposables`.

```
public Stream WriteToFile(string path, string text)
{
  var fs = new FileStream(path, FileMode.Open); // Compliant
  var bytes = Encoding.UTF8.GetBytes(text);
  fs.Write(bytes, 0, bytes.Length);
  return fs;
}

public void ReadFromStream(Stream s)
{
  var sr = new StreamReader(s); // Compliant as it would clo
  // ...
}
```

See

- MITRE, CWE-459 - Incomplete Cleanup

Available In:

sonarlint | sonarcloud | sonarqube