QRs

66063: Secure Coding - API Abuse

2200000: Avoid local variables shadowing class fields

AssociatedValueName: Number of violation occurrences

Description: This rule will check whether local variables are shadowing class fields. In case of C#, classes as well as structs are considered. In case of Visual Basic, modules also considered in addition to classes. In case of classes, only non-private fields of Base classes are considered.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code - Bookmark to field that is being shadowed

Rationale: Both overriding or shadowing a class field (typically non-private) can strongly impact the readability, and therefore the maintainability, of a piece of code.

Reference: CERT, DCL01-C. - Do not reuse variable names in subscopes CERT, DCL51-J. - Do not shadow or obscure identifiers in subscopes

Remediation: Ensure you have an explicit way, usually in form of naming conventions, to name your local variable to avoid conflict with class fields.

RemediationSample:

```
// Field Shadowing

class MyVector {
  private int val = 1;
  private void doLogic() {
    int newValue;
    //...
  }
}

// Variable Shadowing

class MyVector {
  private void doLogic() {
    for (int i = 0; i < 10; i++) {/* ... */}
    for (int i = 0; i < 20; i++) {/* ... */}
  }
}
```

Sample:

```
// Field Shadowing

class MyVector {
  private int val = 1;
  private void doLogic() {
    int val;
    //...
  }
}

// Variable shadowing
```

```
class MyVector {
  private int i = 0;
  private void doLogic() {
    for (i = 0; i < 10; i++) {/* ... */}
    for (int i = 0; i < 20; i++) {/* ... */}
  }
}
```

Total: Number of Methods

2200002: Child class fields should not shadow parent class fields

AssociatedValueName: Number of violation occurrences

Description: This QR will check whether child class fields shadow parent class fields. The check is irrespective of field type and case-sensitivity.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Having a variable with the same name in two unrelated classes is fine, but do the same thing within a class hierarchy and you'll get confusion at best, chaos at worst.

Reference: https://rules.sonarsource.com/csharp/RSPEC-2387

RemediationSample:

```
public class Fruit
{
  protected Season ripe;
  protected Color flesh;

  // ...
}

public class Raspberry : Fruit
{
  private bool ripened;
  private static Color FLESH_COLOR;
}
```

Sample:

```
public class Fruit
{
  protected Season ripe;
  protected Color flesh;

  // ...
}

public class Raspberry : Fruit
{
  private bool ripe; // Noncompliant
  private static Color FLESH; // Noncompliant
}
```

2200004: Inherited member visibility should not be decreased

AssociatedValueName: Number of violation occurrences

Description: This rule raises an issue when a private method in an unsealed type has a signature that is identical to a public method declared in a base type.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Changing an inherited member to private will not prevent access to the base class implementation

Reference: https://rules.sonarsource.com/csharp/RSPEC-4015

RemediationSample:

```
using System;

namespace MyLibrary
{
  public class Foo
  {
    public void SomeMethod(int count) { }
  }
  public sealed class Bar : Foo
  {
```

```
    private void SomeMethod(int count) { }
  }
}
```

Sample:

```
using System;

namespace MyLibrary
{
  public class Foo
  {
    public void SomeMethod(int count) { }
  }
  public class Bar:Foo
  {
    private void SomeMethod(int count) { } // Noncompliant
  }
}
```

2200006: Track "FIXME" tags

AssociatedValueName: Number of violation occurrences

Description: This rule will check the use of FIXME tags in comment for method and classes. All comments, single line and multi-line, are considered that have "FIXME" (case insensitive) at the start of the comment.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: FIXME tags are commonly used to mark places where a bug is suspected, but which the developer wants to deal with later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed.

Reference: MITRE, CWE-546 - Suspicious Comment

Remediation: Fix the issues in code and remove "FIXME" tags.

Sample:

```
private int Divide(int numerator, int denominator)
{
    return numerator / denominator;  // FIXME denominator value might be  0
}
```

Total: Number of Artifacts

2200008: Track "TODO" tags

AssociatedValueName: Number of violation occurrences

Description: This rule will check the use of TODO tags in comment for method and classes. All comments, single line and multi-line, are considered that have "TODO" (case insensitive) at the start of the comment.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed.

Reference: MITRE, CWE-546 - Suspicious Comment

Remediation: Complete remaining tasks and remove "TODO" tags.

Sample:

```
private void DoSomething()
{
  // TODO
}
```

Total: Number of Artifacts

2200010: Classes implementing "IEquatable<T>" should be sealed

AssociatedValueName: Number of violation occurrences

Description: This rule raises an issue when a unsealed, public or protected class implements IEquatable<T> and the Equals is neither virtual nor abstract.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: When a class implements the IEquatable<T> interface, it enters a contract that, in effect, states "I know how to compare two instances of type T or any type derived from T for equality.". However if that class is derived, it is very unlikely that the base class will know how to make a meaningful comparison. Therefore that implicit contract is now broken. Alternatively IEqualityComparer<T> provides a safer interface and is used by collections or Equals could be made virtual.

Reference: https://msdn.microsoft.com/en-us/library/ms132151(v=vs.110).aspx https://rules.sonarsource.com/csharp/RSPEC-4035

Remediation: Make class sealed or use IEqualityComparer<T> instead.

RemediationSample:

```
using System;

namespace MyLibrary
{
    public sealed class Foo : IEquatable<Foo>
    {
        public bool Equals(Foo other)
        {
            // Your code here
        }
    }
}
```

Sample:

```
using System;

namespace MyLibrary
{
  class Base : IEquatable<Base> // Noncompliant
  {
    bool Equals(Base other)
    {
      if (other == null) { return false };
      // do comparison of base properties
    }

    override bool Equals(object other)  => Equals(other as Base);
  }

  class A : Base
  {
    bool Equals(A other)
    {
      if (other == null) { return false };
      // do comparison of A properties
      return base.Equals(other);
    }

    override bool Equals(object other)  => Equals(other as A);
  }

  class B : Base
  {
    bool Equals(B other)
    {
      if (other == null) { return false };
      // do comparison of B properties
     return base.Equals(other);
    }

    override bool Equals(object other)  => Equals(other as B);
  }

  static void Main() {
    A a = new A();
    B b = new B();

    Console.WriteLine(a.Equals(b)); // This calls the WRONG equals. This causes Base::Equals(Base)
    //  to be called which only compares the properties in Base and ignores the fact that
    // a and b are different types. In the working example A::Equals(Object) would have been
    // called and Equals would return false because it correctly recognizes that a and b are
    // different types. If a and b have the same base properties they will  be returned as equal.
  }
}
```

Total: Number of Classes

2200012: Empty arrays and collections should be returned instead of null

AssociatedValueName: Number of violation occurrences

Description: This rule will verify that methods\properties that return arrays\collections do not return null.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Returning null instead of an actual array or collection forces callers of the method to explicitly test for nullity, making them more complex and less readable. Moreover, in many cases, null is used as a synonym for empty.

Reference: CERT, MSC19-C. - For functions that return an array, prefer returning an empty array over a null value CERT, MET55-J. - Return an empty array or collection instead of a null value for methods that return an array or collection

Remediation: Return empty array\collection.

RemediationSample:

```
public Result[] GetResults()
{
    return new Result[0];
}

public IEnumerable<Result> GetResults()
{
    return Enumerable.Empty<Result>();
}

public IEnumerable<Result> GetResults() => Enumerable.Empty<Result>();

public IEnumerable<Result> Results
{
    get
    {
        return Enumerable.Empty<Result>();
    }
}

public IEnumerable<Result> Results => Enumerable.Empty<Result>();
```

Sample:

```
public Result[] GetResults()
{
    return null; // Noncompliant
}

public IEnumerable<Result> GetResults()
{
    return null; // Noncompliant
}

public IEnumerable<Result> GetResults() => null; // Noncompliant

public IEnumerable<Result> Results
{
    get
    {
        return null; // Noncompliant
    }
}

public IEnumerable<Result> Results => null; // Noncompliant
```

Total: Number of Artifacts

2200014: Interface instances should not be cast to concrete types

AssociatedValueName: Number of violation occurrences

Description: This rule will check whether variable of interface type is converted into concrete type. Struct and Class are considered as Concrete type. Note: Abstract classes are not considered as concrete classes.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Needing to cast from an interface to a concrete type indicates that something is wrong with the abstraction in use, likely that something is missing from the interface. Instead of casting to a discrete type, the missing functionality should be added to the interface. Otherwise there is a risk of runtime exceptions.

Remediation: Remove the cast.

Sample:

```
public interface IMyInterface
{
  void DoStuff();
}

public class MyClass1 : IMyInterface
{
  public int Data { get { return new Random().Next(); } }

  public void DoStuff()
  {
    // TODO...
  }
}

public static class DowncastExampleProgram
{
  static void EntryPoint(IMyInterface interfaceRef)
  {
    MyClass1 class1 = (MyClass1)interfaceRef;  // Noncompliant
    int privateData = class1.Data;

    class1 = interfaceRef as MyClass1;  // Noncompliant
    if (class1 != null)
    {
      // ...
    }
  }
}
```

Total: Number of Artifacts

2200016: Ensure proper arguments to Events

AssociatedValueName: Number of violation occurrences

Description: The rule will raise a violation in case of event raising when: 1. NULL is passed as sender when raising an non-static event 2. NULL is passed as event data when raising an event

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: With respect to guidelines from MSDN the following rules must be followed when raising events: 1. DO NOT pass null as the event data parameter when raising an event. 2. DO NOT pass null as the sender when raising a non-static event. It prevents a null reference exception should a method try and do something with the arguments.

Reference: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/event?redirectedfrom=MSDN

Remediation: You should pass EventArgs.Empty if you don’t want to pass any data to the event-handling method. Developers expect this parameter not to be null.

RemediationSample:

```
class AClass {
   public event EventHandler foo;

   protected virtual void OnTfoo(EventArgs e)
   {
       foo?.Invoke(this, e); // Compliant
   }
}
```

Sample:

```
class AClass {
   public event EventHandler foo;

   protected virtual void OnTfoo(EventArgs e)
   {
       foo?.Invoke(null, e); // Noncompliant
   }
}
```

Total: Number of methods

2200018: Avoid using Assembly.LoadFrom, Assembly.LoadFile and Assembly.LoadWithPartialName

AssociatedValueName: Number of violation occurrences

Description: This rule will check the use of Assembly.LoadFrom, Assembly.LoadFile and Assembly.LoadWithPartialName methods

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: The trust level of an assembly that is loaded by using this method is the same as the trust level of the calling assembly. To load an assembly from a byte array with the trust level of the application domain, use the Load(Byte[], Byte[], SecurityContextSource) method.

Reference: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.loadfrom?view=netcore-3.1 https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.loadfile?view=netcore-3.1 https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.loadwithpartialname?view=netcore-3.1

Remediation: Always use Assembly.Load as main method to load DLL.

RemediationSample:

```
static void Main(string[] args)
{
    Assembly.Load(...); // NO VIOLATION
}
```

Sample:

```
static void Main(string[] args)
{
    Assembly.LoadFrom(...); // VIOLATION
    Assembly.LoadFile(...); // VIOLATION
    Assembly.LoadWithPartialName(...); // VIOLATION
}
```

Total: Number of methods and fields and properties initialized using a lambda function

2200020: Avoid methods named without following synchronous/asynchronous convention

AssociatedValueName: Number of violation occurrences

Description: This rule will check if synchronous task could be distinguished as Async or Sync based on name i.e. if async/sync suffixes are used in such methods as expected.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: According to the Task-based Asynchronous Pattern (TAP), methods returning either a System.Threading.Tasks.Task or a System.Threading.Tasks.Task<TResult> are considered asynchronous. Such methods should use the Async suffix. Conversely methods which do not return such Tasks should not have an "Async" suffix.

Reference: https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap

Remediation: Ensure your methods name are following synchronous/asynchronous convention.

RemediationSample:

```
using System.Threading;
using System.Threading.Tasks;

namespace theLibrary
{

  public class theClass
  {
    public Task ReadAsync(byte [] buffer, int offset, int count, CancellationToken cancellationToken) // fixed violation
    {
       // source code
    }

    public int Read() { // fixed violation
       return 0;
    }

  }
}
```

Sample:

```
using System.Threading;
using System.Threading.Tasks;

namespace theLibrary
{

  public class theClass
  {
    public Task Read(byte [] buffer, int offset, int count, CancellationToken cancellationToken)  // violation
    {
```

```
        // source code
    }

    public int ReadAsync() { // violation
        return 0;
    }
  }
}
```

Total: Number of methods

2200022: Culture Dependent String operations should specify culture

Description: This rule will raise violations if string.ToLower(), ToUpper, IndexOf, LastIndexOf, and Compare do not specify culture argument or CompareTo is called.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Calls without a culture may work fine in the system's "home" environment, but break in ways that are extremely difficult to diagnose for customers who use different encodings. Such bugs can be nearly, if not completely, impossible to reproduce when it's time to fix them.

Reference: https://wiki.sei.cmu.edu/confluence/display/java/STR02-J.+Specify+an+appropriate+locale+when+comparing+locale-dependent+data

Remediation: Use Culture argument or use culture invariant version. In case of CompareTo, CompareOrdinal, or Compare with culture.

RemediationSample:

```
var lowered = someString.ToLower(CultureInfo.InvariantCulture);

-or-

var lowered = someString.ToLowerInvariant();
```

Sample:

```
var lowered = someString.ToLower(); //Noncompliant
```

Total: Number of Artifacts

2200024: Mutable static fields of type System.Collections.Generic.ICollection<T> or System.Array should not be public static

Description: This rule checks for fields that are public static of type System.Array or System.Collections.Generic.ICollection<T> and are not read-only.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: If field is static and public and not read-only, it can affect every class that uses them. This can lead to unexpected behavior.

Remediation: Make visibility of fields protected\private or make them of type Immutable\read-only. This can done through: - make fields read-only (with inline initialization) - is of type System.Collections.ObjectModel.ReadOnlyCollection<T> System.Collections.ObjectModel.ReadOnlyDictionary<TKey, TValue> System.Collections.Immutable.IImmutableArray<T> System.Collections.Immutable.IImmutableDictionary<TKey, TValue> System.Collections.Immutable.IImmutableList<T> System.Collections.Immutable.IImmutableSet<T> System.Collections.Immutable.IImmutableStack<T> System.Collections.Immutable.IImmutableQueue<T>

RemediationSample:

```
public class A
{
  protected static string[] strings1 = {"first","second"};
  protected static List<String> strings3 = new List<String>();
  // ...
}
```

Sample:

```
public class A
{
  public static string[] strings1 = {"first","second"};  // Noncompliant
  public static List<String> strings3 = new List<String>();  // Noncompliant
  // ...
}
```

Total: Number of Artifacts

2200026: Avoid creating exception without throwing them

Description: This rule will check whether an exception type object is created but not thrown.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Only creating exception and throwing it would mean that either it is a mistake or it is used for side effect of object creation.

Remediation: Throw the exception or remove the statement

RemediationSample:

```
var o = new Exception();

throw o;

throw new Exception();
```

Sample:

```
var e = new Exception();

new Exception();
```

Total: Number of Artifacts

2200028: Use Logical OR instead of Bitwise OR in boolean context

Description: This rule will check whether bitwise OR (|) is used instead of Logical OR (||) in boolean context.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: When Bitwise OR is used instead of Logical OR in boolean context, it is most probably a mistake or it is intended for side effect which is incorrect programming practice.

Remediation: Use Logical OR

RemediationSample:

```
Class AClass {
      private int Return1() {
         return 1;
      }

      private int Return0() {
         return 0;
      }

      public void Test() {
         bool b1 = false;
         bool b2 = true;
         var x = b1 || b2;
         x = ReturnFalse() || ReturnTrue();
     }
}
```

Sample:

```
Class AClass {
      private int Return1() {
         return 1;
      }

      private int Return0() {
         return 0;
      }

      public void Test() {
         bool b1 = false;
         bool b2 = true;
         var x = b1 | b2;
         x = ReturnFalse() | ReturnTrue();
     }
}
```

Total: Number of Artifacts

2200030: Avoid empty finalizers

Description: The rule will raise a violation when a type implements a finalizer that is empty. Even finalizer with only statement with calls to Debug.Fail and it is not in #if DEBUG part, it will be considered as violation since Debug.Fail is omitted for non-DEBUG configuration.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Whenever you can, avoid finalizers because of the additional performance overhead that's involved in tracking object lifetime. The garbage collector runs the finalizer before it collects the object. This means that at least two collections are required to collect the object. An empty finalizer incurs this added overhead without any benefit.

Reference: https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca1821 https://cwe.mitre.org/data/definitions/1069.html

Remediation: Avoid using empty finalizers

Sample:

```
public class Class1
    {
        // Violation occurs because the finalizer is empty.
        ~Class1()
        {
        }
    }

    public class Class2
    {
        // Violation occurs because Debug.Fail is a conditional method.
        // The finalizer will contain code only if the DEBUG directive
        // symbol is present at compile time. When the DEBUG
        // directive is not present, the finalizer will still exist, but
        // it will be empty.
        ~Class2()
        {
            Debug.Fail("Finalizer called!");
        }
    }
```

Total: Number of finalizers

2200032: Avoid recursive type inheritance

Description: This rule will raise violation if Recursion is used in type inheritance.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Unlike methods, it is not possible to break out of recursion. If used, it will fail at runtime.

Remediation: Do not use recursive type inheritance.

Sample:

```
class C1<T> {
      }

      class C2KO<S> : C1<C2KO<C1<S>>> // Noncompliant
      {
         public int x = 101;
      }

      class C3KO<S> : C1<C3KO<C3KO<S>>> // Noncompliant
      {
         public int x = 101;
      }
```

Total: Number pf classes

2200034: For loop stop condition should be invariant

AssociatedValueName: Number of violation occurrences

Description: This rule will give violation if condition in for loop is not invariant.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: A for loop stop condition should test the loop counter against an invariant value (i.e. one that is true at both the beginning and ending of every loop iteration). Ideally, this means that the stop condition is set to a local variable just before the loop begins. Stop conditions that are not invariant are slightly less efficient, as well as being difficult to understand and maintain, and likely lead to the introduction of errors in the future. This gives violation when the loop counters are updated in the body of the for loop.

Remediation: Make for loop condition invariant.

RemediationSample:

```
class Foo
{
    static void Main()
    {
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

Sample:

```
class Foo
{
    static void Main()
    {
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine(i);
            if (condition)
            {
               i = 20;
            }
        }
    }
}
```

Total: Number of methods

2200036: Ensure constructors of serializable classes are secure

Description: This rule raises an issue when a type implements the System.Runtime.Serialization.ISerializable interface, is not a delegate or interface, is declared in an assembly that allows partially trusted callers and has a constructor that takes a System.Runtime.Serialization.SerializationInfo object and a System.Runtime.Serialization.StreamingContext object which is not secured by a security check, but one or more of the regular constructors in the type is secured.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Because serialization constructors allocate and initialize objects, security checks that are present on regular constructors must also be present on a serialization constructor. Failure to do so would allow callers that could not otherwise create an instance to use the serialization constructor to do this.

Reference: https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization.html

Remediation: Make constructors of serializable classes secure.

RemediationSample:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Security;
using System.Security.Permissions;

[assembly: AllowPartiallyTrustedCallersAttribute()]
namespace MyLibrary
{
    [Serializable]
    public class Foo : ISerializable
    {
        private int n;

        [FileIOPermissionAttribute(SecurityAction.Demand, Unrestricted = true)]
        public Foo()
        {
           n = -1;
        }

        [FileIOPermissionAttribute(SecurityAction.Demand, Unrestricted = true)]
        protected Foo(SerializationInfo info, StreamingContext context)
        {
           n = (int)info.GetValue("n", typeof(int));
        }

        void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
        {
           info.AddValue("n", n);
        }
    }
}
```

Sample:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Security;
using System.Security.Permissions;

[assembly: AllowPartiallyTrustedCallersAttribute()]
namespace MyLibrary
{
    [Serializable]
    public class Foo : ISerializable
    {
        private int n;

        [FileIOPermissionAttribute(SecurityAction.Demand, Unrestricted = true)]
        public Foo()
        {
           n = -1;
        }

        protected Foo(SerializationInfo info, StreamingContext context) // Noncompliant
        {
           n = (int)info.GetValue("n", typeof(int));
        }

        void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
        {
           info.AddValue("n", n);
        }
    }
}
```

Total: Number of constructors

2200038: Merge adjacent try blocks with identical catch/finally statements

Description: This rule will raise a violation when adjacent try-catch blocks have identical catch blocks.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Adjacent try-catch blocks having identical catch blocks must be merged to improve readability of code.

Remediation: Merge the try-catch blocks.

RemediationSample:

```
try
{
  DoTheFirstThing(a, b);
  DoSomeOtherStuff();
  DoTheSecondThing();
}
catch (InvalidOperationException ex)
{
  HandleException(ex);
}

try  // Compliant; catch handles exception differently
{
  DoTheThirdThing(a);
}
catch (InvalidOperationException ex)
{
  LogAndDie(ex);
}
```

Sample:

```
try
{
  DoTheFirstThing(a, b);
}
catch (InvalidOperationException ex)
{
  HandleException(ex);
}

DoSomeOtherStuff();

try  // Noncompliant; catch is identical to previous
{
  DoTheSecondThing();
}
catch (InvalidOperationException ex)
{
  HandleException(ex);
}

try  // Compliant; catch handles exception differently
{
```

```
  DoTheThirdThing(a);
}
catch (InvalidOperationException ex)
{
  LogAndDie(ex);
}
```

Total: Number of methods

2200040: Avoid assignments in sub-expressions

AssociatedValueName: Number of violation occurrences

Description: This rule will check if assignments are done in if\switch\method\constructor calls.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Assignments within sub-expressions are hard to spot and therefore make the code less readable. Ideally, sub-expressions should not have side-effects.

Reference: MITRE, CWE-481 - Assigning instead of Comparing CERT, EXP45-C. - Do not perform assignments in selection statements CERT, EXP51-J. - Do not perform assignments in conditional expressions

Remediation: Remove assignments from if\switch\method calls\constructor calls sub-expressions.

RemediationSample:

```
var result = str.Substring(index, length);
if (string.IsNullOrEmpty(result))
{
  //...
}
```

Sample:

```
if (string.IsNullOrEmpty(result = str.Substring(index, length))) // Noncompliant
{
  //...
}
```

Total: Number of Artifacts

2200042: Avoid creating new instance of shared instance (.NET)

Description: This rule will raise a violation upon invocation of a constructor of a class considered as shared with [PartCreationPolicyAttribute]

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: If a class is marked such that only a single object of the class will be exported as a shared object [PartCreationPolicy(CreationPolicy.Shared)], then invoking the constructor and creating new instances with it will result in unexpected behavior.

Remediation: Prefer using the created instance and its resources.

RemediationSample:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ComponentModel.Composition;
using System.ComponentModel.Design;
using Microsoft.Extensions.DependencyInjection;

namespace GenericObject {

   interface IInterface {

   }

   [PartCreationPolicy(CreationPolicy.Shared)]
   class AService : IInterface {
      public AService() {
         System.Console.WriteLine(System.Reflection.MethodBase.GetCurrentMethod().Name);
      }
   }

   class AServiceUser {
```

```
      private ServiceContainer _serviceContainer;
      public AServiceUser() {
         _serviceContainer = new ServiceContainer();
         _serviceContainer.AddService(typeof(IInterface), new AService());
         UseAService();
      }

      public void UseAService() {
         var aservice = _serviceContainer.GetService(typeof(IInterface)); //VIOLATION FIXED
      }
   }
}
```

Sample:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ComponentModel.Composition;
using System.ComponentModel.Design;
using Microsoft.Extensions.DependencyInjection;

namespace GenericObject {

   interface IInterface {

   }

   [PartCreationPolicy(CreationPolicy.Shared)]
   class AService : IInterface {
      public AService() {
         System.Console.WriteLine(System.Reflection.MethodBase.GetCurrentMethod().Name);
      }
   }

   class AServiceUser {
      private ServiceContainer _serviceContainer;
      public AServiceUser() {
         _serviceContainer = new ServiceContainer();
         _serviceContainer.AddService(typeof(IInterface), new AService());
         UseAService();
      }

      public void UseAService() {
         var aservice = new AService(); //VIOLATION
      }
   }
}
```

Total: Number of Artifacts

2200044: Recursion should not be infinite

Description: If recursive methods have a call that results in them being recursive from every control path, recursion becomes infinite. This QR checks for such methods.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for infinite recursive method

Rationale: Recursion if become infinite, would crash the program.

Remediation: Change code to make sure control is returned to caller.

RemediationSample:

```
void RecursiveMethod(int i) {
   if (100 < i) {
      RecursiveMethod(++i);
   }
}
```

Sample:

```
void RecursiveMethod(int i) {
   RecursiveMethod(++i);
}
```

Total: Number of methods

2200046: Ensure Serializable Types Follow Best Practices

Description: This rule will raise a violation when an externally visible type is assignable to the System.Runtime.Serialization.ISerializable interface and one of the following conditions is true: - The System.SerializableAttribute attribute is missing. - Non-serializable fields are not marked with the System.NonSerializedAttribute attribute. - There is no serialization constructor. - An unsealed type has a serialization constructor that is not protected. - A sealed type has a serialization constructor that is not private. - An unsealed type has a ISerializable.GetObjectData that is not both public and virtual. - A derived type has a serialization constructor that does not call the base constructor. - A derived type has a ISerializable.GetObjectData method that does not

call the base method. - A derived type has serializable fields but the ISerializable.GetObjectData method is not overridden.

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Instance fields that are declared in a type that inherits the System.Runtime.Serialization.ISerializable interface are not automatically included in the serialization process. To include the fields, the type must implement the GetObjectData method and the serialization constructor. If the fields should not be serialized, apply the NonSerializedAttribute attribute to the fields to explicitly indicate the decision. In types that are not sealed, implementations of the GetObjectData method should be externally visible. Therefore, the method can be called by derived types, and is overridable.

Reference: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.iserializable?view=net-5.0 https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute?view=net-5.0 https://medium.com/@CPP_Coder/how-to-not-shoot-yourself-in-the-foot-when-working-with-serialization-20a9a13b69b5

Remediation: Follow best practices for serializable type.

Total: Number of Artifacts

2200048: Members of larger scope element should not have conflicting transparency annotations

AssociatedValueName: Number of violation occurrences

Description: This rule will raise a violation when a type member is marked with a System.Security security attribute that has a different transparency than the security attribute of a container of the member. Following Security Attributes are considered for comparison: System.Security.SecurityCriticalAttribute System.Security.SecurityRulesAttribute System.Security.SecuritySafeCriticalAttribute System.Security.SecurityTransparentAttribute System.Security.SecurityTreatAsSafeAttribute System.Security.SuppressUnmanagedCodeSecurityAttribute System.Security.UnverifiableCodeAttribute

Output: Associated to each violation, the following information is provided: - The number of violation occurrences - Bookmarks for violation occurrences found in the source code

Rationale: Transparency attributes are applied from code elements of larger scope to elements of smaller scope. The transparency attributes of code elements with larger scope take precedence over transparency attributes of code elements that are contained in the first element. For example, a class that is marked with the SecurityCriticalAttribute attribute cannot contain a method that is marked with the SecuritySafeCriticalAttribute attribute.

Reference: https://docs.microsoft.com/en-us/visualstudio/code-quality/ca2136?view=vs-2019 OWASP Top 10 2017 Category A6 - Security Misconfiguration

Remediation: To fix this violation, remove the security attribute from the code element that has lower scope, or change its attribute to be the same as the containing code element.

RemediationSample:

```
using System;
using System.Security;

namespace TransparencyWarningsDemo
{

    [SecurityCritical]
    public class CriticalClass
    {
       //Violation Fixed
        public void SafeCriticalMethod()
        {
        }
    }
}
```

Sample:

```
using System;
using System.Security;

namespace TransparencyWarningsDemo
{

    [SecurityCritical]
    public class CriticalClass
    {
        // CA2136 violation - this method is not really safe critical, since the larger scoped type annotation
        // has precidence over the smaller scoped method annotation.  This can be fixed by removing the
        // SecuritySafeCritical attribute on this method
        [SecuritySafeCritical] //Violation
        public void SafeCriticalMethod()
        {
        }
    }
}
```

Total: Number of Artifacts