Contents

- Code analysis documentation
- Analyze managed code
  - Overview
  - Source code analysis
    - Overview
    - Install source code analyzers
    - Configure source code analyzers
    - Live code analysis scope
    - Run code analysis manually
    - Disable code analysis
    - FAQ
    - FxCop analyzers
      - Install FxCop analyzers
      - Migrate to FxCop analyzers
        - Migrate from legacy analysis to FxCop analyzers
        - FAQ
        - Ported rules
      - Configure FxCop analyzers
        - Configure rules and projects
        - Scope options
        - Enable a category of rules
  - Legacy analysis
    - Overview
    - Legacy analysis walkthrough
    - Migrate to FxCop analyzers
    - Run legacy analysis manually
    - Configuration
      - Configure legacy analysis
      - Enable automatic legacy code analysis

Customize the legacy code analysis dictionary
Suppression
Suppress legacy analysis warnings for generated code
Legacy analysis application errors
Legacy analysis policy errors
Overview
CA0001
CA0051
CA0052
CA0053
CA0054
CA0055
CA0056
CA0057
CA0058
CA0059
CA0060
CA0061
CA0062
CA0063
CA0064
A0065
CA0066
CA0067
CA0068
CA0069
CA0070
CA0501
CA0502
CA0503
CA0504
CA0505

- FxCopCmd errors
- Check-in policies
  - How to create or update check-in policies
  - Implement legacy analysis check-in policies
  - Enforce maintainable code
  - Synchronize project rule sets with check-in policy
  - Version compatibility
- Rule sets
  - Overview
  - Configure rule sets
  - Create a custom rule set
  - Use the rule set editor
- Code metrics
  - Overview
  - Generate code metrics data
  - Work with code metrics data
- Reference
  - Rule sets
    - Overview
    - All Rules
    - Basic Correctness Rules
    - Basic Design Guideline Rules
    - Extended Correctness Rules
    - Extended Design Guidelines Rules
    - Globalization Rules
    - Managed Minimum Rules
    - Managed Recommended Rules
    - Mixed Minimum Rules
    - Mixed Recommended Rules
    - Native Minimum Rules
    - Native Recommended Rules
    - Security Rules

Suppress warnings
Managed code warnings
  Overview
  Warnings by CheckId
  Cryptography warnings
    Overview
    CA5350: Do Not Use Weak Cryptographic Algorithms
    CA5351 Do Not Use Broken Cryptographic Algorithms
  Design warnings
    Overview
    CA1000: Do not declare static members on generic types
    CA1001: Types that own disposable fields should be disposable
    CA1002: Do not expose generic lists
    CA1003: Use generic event handler instances
    CA1004: Generic methods should provide type parameter
    CA1005: Avoid excessive parameters on generic types
    CA1006: Do not nest generic types in member signatures
    CA1007: Use generics where appropriate
    CA1008: Enums should have zero value
    CA1009: Declare event handlers correctly
    CA1010: Collections should implement generic interface
    CA1011: Consider passing base types as parameters
    CA1012: Abstract types should not have constructors
    CA1013: Overload operator equals on overloading add and subtract
    CA1014: Mark assemblies with CLSCompliantAttribute
    CA1016: Mark assemblies with AssemblyVersionAttribute
    CA1017: Mark assemblies with ComVisibleAttribute
    CA1018: Mark attributes with AttributeUsageAttribute
    CA1019: Define accessors for attribute arguments
    CA1020: Avoid namespaces with few types
    CA1021: Avoid out parameters
    CA1023: Indexers should not be multidimensional

CA1024: Use properties where appropriate

CA1025: Replace repetitive arguments with params array

CA1026: Default parameters should not be used

CA1027: Mark enums with FlagsAttribute

CA1028: Enum storage should be Int32

CA1030: Use events where appropriate

CA1031: Do not catch general exception types

CA1032: Implement standard exception constructors

CA1033: Interface methods should be callable by child types

CA1034: Nested types should not be visible

CA1035: ICollection implementations have strongly typed members

CA1036: Override methods on comparable types

CA1038: Enumerators should be strongly typed

CA1039: Lists are strongly typed

CA1040: Avoid empty interfaces

CA1041: Provide ObsoleteAttribute message

CA1043: Use integral or string argument for indexers

CA1044: Properties should not be write only

CA1045: Do not pass types by reference

CA1046: Do not overload operator equals on reference types

CA1047: Do not declare protected members in sealed types

CA1048: Do not declare virtual members in sealed types

CA1049: Types that own native resources should be disposable

CA1050: Declare types in namespaces

CA1051: Do not declare visible instance fields

CA1052: Static holder types should be sealed

CA1053: Static holder types should not have constructors

CA1054: URI parameters should not be strings

CA1055: URI return values should not be strings

CA1056: URI properties should not be strings

CA1057: String URI overloads call System.Uri overloads

CA1058: Types should not extend certain base types

CA1059: Members should not expose certain concrete types
CA1060: Move P-Invokes to NativeMethods class
CA1061: Do not hide base class methods
CA1062: Validate arguments of public methods
CA1063: Implement IDisposable correctly
CA1064: Exceptions should be public
CA1065: Do not raise exceptions in unexpected locations
CA1066: Implement IEquatable when overriding Equals
CA1067: Override Equals when implementing IEquatable
CA1068: CancellationToken parameters must come last
CA1069: Enums should not have duplicate values
CA1070: Do not declare event fields as virtual
CA2210: Assemblies should have valid strong names
Documentation warnings
Overview
CA1200: Avoid using cref tags with a prefix
Globalization warnings
Overview
CA1300: Specify MessageBoxOptions
CA1301: Avoid duplicate accelerators
CA1302: Do not hardcode locale specific strings
CA1303: Do not pass literals as localized parameters
CA1304: Specify CultureInfo
CA1305: Specify IFormatProvider
CA1306: Set locale for data types
CA1307: Specify StringComparison
CA1308: Normalize strings to uppercase
CA1309: Use ordinal StringComparison
CA2101: Specify marshaling for P-Invoke string arguments
Interoperability warnings
Overview
CA1400: P-Invoke entry points should exist

CA1401: P-Invokes should not be visible
CA1402: Avoid overloads in COM visible interfaces
CA1403: Auto layout types should not be COM visible
CA1404: Call GetLastError immediately after P-Invoke
CA1405: COM visible type base types should be COM visible
CA1406: Avoid Int64 arguments for Visual Basic 6 clients
CA1407: Avoid static members in COM visible types
CA1408: Do not use AutoDual ClassInterfaceType
CA1409: Com visible types should be creatable
CA1410: COM registration methods should be matched
CA1411: COM registration methods should not be visible
CA1412: Mark ComSource Interfaces as IDispatch
CA1413: Avoid non-public fields in COM visible value types
CA1414: Mark boolean P-Invoke arguments with MarshalAs
CA1415: Declare P-Invokes correctly
CA1417: Do not use OutAttribute on string parameters for P/Invokes
Maintainability warnings
Overview
CA1500: Variable names should not match field names
CA1501: Avoid excessive inheritance
CA1502: Avoid excessive complexity
CA1504: Review misleading field names
CA1505: Avoid unmaintainable code
CA1506: Avoid excessive class coupling
CA1507: Use nameof in place of string
CA1508: Avoid dead conditional code
CA1509: Invalid entry in code metrics configuration file
Mobility warnings
Overview
CA1600: Do not use idle process priority
CA1601: Do not use timers that prevent power state changes
Naming warnings

Overview
CA1700: Do not name enum values 'Reserved'
CA1701: Resource string compound words should be cased correctly
CA1702: Compound words should be cased correctly
CA1703: Resource strings should be spelled correctly
CA1704: Identifiers should be spelled correctly
CA1707: Identifiers should not contain underscores
CA1708: Identifiers should differ by more than case
CA1709: Identifiers should be cased correctly
CA1710: Identifiers should have correct suffix
CA1711: Identifiers should not have incorrect suffix
CA1712: Do not prefix enum values with type name
CA1713: Events should not have before or after prefix
CA1714: Flags enums should have plural names
CA1715: Identifiers should have correct prefix
CA1716: Identifiers should not match keywords
CA1717: Only FlagsAttribute enums should have plural names
CA1719: Parameter names should not match member names
CA1720: Identifiers should not contain type names
CA1721: Property names should not match get methods
CA1722: Identifiers should not have incorrect prefix
CA1725: Parameter names should match base declaration
CA1724: Type Names Should Not Match Namespaces
CA1726: Use preferred terms
Performance warnings
Overview
CA1800: Do not cast unnecessarily
CA1801: Review unused parameters
CA1802: Use Literals Where Appropriate
CA1804: Remove unused locals
CA1805: Do not initialize unnecessarily
CA1806: Do not ignore method results

CA1809: Avoid excessive locals
CA1810: Initialize reference type static fields inline
CA1811: Avoid uncalled private code
CA1812: Avoid uninstantiated internal classes
CA1813: Avoid unsealed attributes
CA1814: Prefer jagged arrays over multidimensional
CA1815: Override equals and operator equals on value types
CA1816: Call GC.SuppressFinalize correctly
CA1819: Properties should not return arrays
CA1820: Test for empty strings using string length
CA1821: Remove empty finalizers
CA1822: Mark members as static
CA1823: Avoid unused private fields
CA1824: Mark assemblies with NeutralResourcesLanguageAttribute
CA1825: Avoid zero-length array allocations
CA1826: Use property instead of Linq Enumerable method
CA1827: Do not use Count/LongCount when Any can be used
CA1828: Do not use CountAsync/LongCountAsync when AnyAsync can be used
CA1829: Use Length/Count property instead of Enumerable.Count method
CA1830: Prefer strongly-typed Append and Insert method overloads on StringBuilder
CA1831: Use AsSpan instead of Range-based indexers for string when appropriate
CA1832: Use AsSpan or AsMemory instead of Range-based indexers for getting ReadOnlySpan or ReadOnlyMemory portion of an array
CA1833: Use AsSpan or AsMemory instead of Range-based indexers for getting Span or Memory portion of an array
CA1835: Prefer the memory overloads of Stream ReadAsync/WriteAsync methods
CA1836: Prefer IsEmpty over Count when available
Portability warnings
Overview
CA1900: Value type fields should be portable
CA1901: P-Invoke declarations should be portable

CA1903: Use only API from targeted framework
Reliability warnings
Overview
CA2000: Dispose objects before losing scope
CA2001: Avoid calling problematic methods
CA2002: Do not lock on objects with weak identity
CA2003: Do not treat fibers as threads
CA2004: Remove calls to GC.KeepAlive
CA2006: Use SafeHandle to encapsulate native resources
CA2007: Do not directly await a Task
CA2009: Do not call ToImmutableCollection on an ImmutableCollection value
CA2011: Do not assign property within its setter
CA2012: Use ValueTasks correctly
CA2013: Do not use ReferenceEquals with value types
CA2014: Do not use stackalloc in loops
CA2015: Do not define finalizers for types derived from MemoryManager<T>
CA2016: Forward the CancellationToken parameter to methods that take one
Security warnings
Overview
CA2100: Review SQL queries for security vulnerabilities
CA2102: Catch non-CLSCompliant exceptions in general handlers
CA2103: Review imperative security
CA2104: Do not declare read only mutable reference types
CA2105: Array fields should not be read only
CA2106: Secure asserts
CA2107: Review deny and permit only usage
CA2108: Review declarative security on value types
CA2109: Review visible event handlers
CA2111: Pointers should not be visible
CA2112: Secured types should not expose fields
CA2114: Method security should be a superset of type
CA2115: Call GC.KeepAlive when using native resources

CA2116: APTCA methods should only call APTCA methods

CA2117: APTCA types should only extend APTCA base types

CA2118: Review SuppressUnmanagedCodeSecurityAttribute usage

CA2119: Seal methods that satisfy private interfaces

CA2120: Secure serialization constructors

CA2121: Static constructors should be private

CA2122: Do not indirectly expose methods with link demands

CA2123: Override link demands should be identical to base

CA2124: Wrap vulnerable finally clauses in outer try

CA2126: Type link demands require inheritance demands

CA2130: Security critical constants should be transparent

CA2131: Security critical types may not participate in type equivalence

CA2132: Default constructors must be at least as critical as base type default constructors

CA2133: Delegates must bind to methods with consistent transparency

CA2134: Methods must keep consistent transparency when overriding base methods

CA2135: Level 2 assemblies should not contain LinkDemands

CA2136: Members should not have conflicting transparency annotations

CA2137: Transparent methods must contain only verifiable IL

CA2138: Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute

CA2139: Transparent methods may not use the HandleProcessCorruptingExceptions attribute

CA2140: Transparent code must not reference security critical items

CA2141:Transparent methods must not satisfy LinkDemands

CA2142: Transparent code should not be protected with LinkDemands

CA2143: Transparent methods should not use security demands

CA2144: Transparent code should not load assemblies from byte arrays

CA2145: Transparent methods should not be decorated with the SuppressUnmanagedCodeSecurityAttribute

CA2146: Types must be at least as critical as their base types and interfaces

CA2147: Transparent methods may not use security asserts

CA2149: Transparent methods must not call into native code

CA2151: Fields with critical types should be security critical

CA5122 P-Invoke declarations should not be safe critical

CA2153: Avoid Handling Corrupted State Exceptions

CA2300: Do not use insecure deserializer BinaryFormatter

CA2301: Do not call BinaryFormatter.Deserialize without first setting BinaryFormatter.Binder

CA2302: Ensure BinaryFormatter.Binder is set before calling BinaryFormatter.Deserialize

CA2305: Do not use insecure deserializer LosFormatter

CA2310: Do not use insecure deserializer NetDataContractSerializer

CA2311: Do not deserialize without first setting NetDataContractSerializer.Binder

CA2312: Ensure NetDataContractSerializer.Binder is set before deserializing

CA2315: Do not use insecure deserializer ObjectStateFormatter

CA2321: Do not deserialize with JavaScriptSerializer using a SimpleTypeResolver

CA2322: Ensure JavaScriptSerializer is not initialized with SimpleTypeResolver before deserializing

CA2326: Do not use TypeNameHandling values other than None

CA2327: Do not use insecure JsonSerializerSettings

CA2328: Ensure that JsonSerializerSettings are secure

CA2329: Do not deserialize with JsonSerializer using an insecure configuration

CA2350: Ensure DataTable.ReadXml()'s input is trusted

CA2351: Ensure DataSet.ReadXml()'s input is trusted

CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks

CA2353: Unsafe DataSet or DataTable in serializable type

CA2354: Unsafe DataSet or DataTable in deserialized object graph can be vulnerable to remote code execution attack

CA2355: Unsafe DataSet or DataTable in deserialized object graph

CA2356: Unsafe DataSet or DataTable type in web deserialized object graph

CA2330: Ensure that JsonSerializer has a secure configuration when deserializing

CA3001: Review code for SQL injection vulnerabilities

CA3002: Review code for XSS vulnerabilities

CA3003: Review code for file path injection vulnerabilities

CA3004: Review code for information disclosure vulnerabilities

CA3005: Review code for LDAP injection vulnerabilities

CA3006: Review code for process command injection vulnerabilities

CA3007: Review code for open redirect vulnerabilities

CA3008: Review code for XPath injection vulnerabilities

CA3009: Review code for XML injection vulnerabilities

CA3010: Review code for XAML injection vulnerabilities

CA3011: Review code for DLL injection vulnerabilities

CA3012: Review code for regex injection vulnerabilities

CA3061: Do not add schema by URL

CA3075: Insecure DTD Processing

CA3076: Insecure XSLT Script Execution

CA3077: Insecure Processing in API Design, XML Document and XML Text Reader

CA3147: Mark verb handlers with ValidateAntiForgeryToken

CA5358: Do Not Use Unsafe Cipher Modes

CA5359: Do not disable certificate validation

CA5360: Do not call dangerous methods in deserialization

CA5361: Do not disable SChannel use of strong crypto

CA5362: Potential reference cycle in deserialized object graph

CA5363: Do not disable request validation

CA5364: Do not use deprecated security protocols

CA5365: Do Not Disable HTTP Header Checking

CA5366: Use XmlReader For DataSet Read XML

CA5367: Do Not Serialize Types With Pointer Fields

CA5368: Set ViewStateUserKey For Classes Derived From Page

CA5369: Use XmlReader for Deserialize

CA5370: Use XmlReader for validating reader

CA5371: Use XmlReader for schema read

CA5372: Use XmlReader for XPathDocument

CA5373: Do not use obsolete key derivation function

CA5374: Do Not Use XslTransform

CA5375: Do not use account shared access signature

CA5376: Use SharedAccessProtocol HttpsOnly

CA5377: Use container level access policy
CA5378: Do not disable ServicePointManagerSecurityProtocols
CA5379: Do not use weak key derivation function algorithm
CA5380: Do not add certificates to root store
CA5381: Ensure certificates are not added to root store
CA5382: Use secure cookies in ASP.NET Core
CA5383: Ensure use secure cookies in ASP.NET Core
CA5384: Do not use digital signature algorithm (DSA)
CA5385: Use Rivest–Shamir–Adleman (RSA) algorithm with sufficient key size
CA5386: Avoid hardcoding SecurityProtocolType value
CA5387: Do not use weak key derivation function with insufficient iteration count
CA5388: Ensure sufficient iteration count when using weak key derivation function
CA5389: Do not add archive item's path to the target file system path
CA5390: Do not hard-code encryption key
CA5391: Use antiforgery tokens in ASP.NET Core MVC controllers
CA5392: Use DefaultDllImportSearchPaths attribute for P/Invokes
CA5393: Do not use unsafe DllImportSearchPath value
CA5394: Do not use insecure randomness
CA5395: Miss HttpVerb attribute for action methods
CA5396: Set HttpOnly to true for HttpCookie
CA5397: Do not use deprecated SslProtocols values
CA5398: Avoid hardcoded SslProtocols values
CA5399: Definitely disable HttpClient certificate revocation list check
CA5400: Ensure HttpClient certificate revocation list check is not disabled
CA5401: Do not use CreateEncryptor with non-default IV
CA5402: Use CreateEncryptor with the default IV
CA5403: Do not hard-code certificate
Usage warnings
Overview
CA1801: Review unused parameters
CA1806: Do not ignore method results
CA1816: Call GC.SuppressFinalize correctly

CA2200: Rethrow to preserve stack details

CA2201: Do not raise reserved exception types

CA2202: Do not dispose objects multiple times

CA2204: Literals should be spelled correctly

CA2205: Use managed equivalents of Win32 API

CA2207: Initialize value type static fields inline

CA2208: Instantiate argument exceptions correctly

CA2211: Non-constant fields should not be visible

CA2212: Do not mark serviced components with WebMethod

CA2213: Disposable fields should be disposed

CA2214: Do not call overridable methods in constructors

CA2215: Dispose methods should call base class dispose

CA2216: Disposable types should declare finalizer

CA2217: Do not mark enums with FlagsAttribute

CA2218: Override GetHashCode on overriding Equals

CA2219: Do not raise exceptions in exception clauses

CA2220: Finalizers should call base class finalizer

CA2221: Finalizers should be protected

CA2222: Do not decrease inherited member visibility

CA2223: Members should differ by more than return type

CA2224: Override equals on overloading operator equals

CA2225: Operator overloads have named alternates

CA2226: Operators should have symmetrical overloads

CA2227: Collection properties should be read only

CA2228: Do not ship unreleased resource formats

CA2229: Implement serialization constructors

CA2230: Use params for variable arguments

CA2231: Overload operator equals on overriding ValueType.Equals

CA2232: Mark Windows Forms entry points with STAThread

CA2233: Operations should not overflow

CA2234: Pass System.Uri objects instead of strings

CA2235: Mark all non-serializable fields

CA2236: Call base class methods on ISerializable types

CA2237: Mark ISerializable types with SerializableAttribute

CA2238: Implement serialization methods correctly

CA2239: Provide deserialization methods for optional fields

CA2240: Implement ISerializable correctly

CA2241: Provide correct arguments to formatting methods

CA2242: Test for NaN correctly

CA2243: Attribute string literals should parse correctly

CA2244: Do not duplicate indexed element initializations

CA2245: Do not assign a property to itself

CA2246: Do not assign a symbol and its member in the same statement

CA2247: Argument passed to TaskCompletionSource constructor should be TaskCreationOptions enum instead of TaskContinuationOptions enum

CA2248: Provide correct enum argument to Enum.HasFlag

Automatic feature suspension

Overview of code analysis for managed code in Visual Studio

2/3/2020 • 2 minutes to read • Edit Online

Visual Studio can perform code analysis of managed code in two ways: with legacy analysis, also known as FxCop static analysis of managed assemblies, and with the more modern .NET Compiler Platform-based code analyzers. .NET Compiler Platform-based code analyzers, which analyze your code live as you type, replace legacy FxCop static code analysis, which only analyzes compiled code.

Overview of source code analyzers

3/16/2020 • 4 minutes to read • Edit Online

.NET Compiler Platform ("Roslyn") code analyzers inspect your C# or Visual Basic code for style, quality and maintainability, design, and other issues.

- Some analyzers are built in to Visual Studio. The diagnostic ID, or code, for these analyzers is of the format IDExxxx, for example, IDE0067. Most of these built-in analyzers inspect code style, and you can configure preferences on the text editor options page or in an EditorConfig file. A handful of built-in analyzers look at code quality.
- You can install additional analyzers as a NuGet package or a Visual Studio extension. For example:
  - FxCop analyzers, Microsoft's recommended code quality analyzers
  - Third party analyzers, such as StyleCop, Roslynator, XUnit Analyzers, and Sonar Analyzer

If rule violations are found by an analyzer, they're reported in the code editor (as a *squiggle* under the offending code) and in the Error List window.

Many analyzer rules, or *diagnostics*, have one or more associated *code fixes* that you can apply to correct the problem. The analyzer diagnostics that are built into Visual Studio each have an associated code fix. Code fixes are shown in the light bulb icon menu along with other types of Quick Actions. For information about these code fixes, see Common Quick Actions.

Source code analysis versus legacy analysis

Source analysis by Roslyn analyzers replaces legacy analysis for managed code. Many of the legacy analysis rules have already been rewritten as Roslyn code analyzers. For newer project templates such as .NET Core and .NET Standard projects, legacy analysis is not even available.

Like legacy analysis rule violations, source code analysis violations appear in the Error List window in Visual Studio. In addition, source code analysis violations also show up in the code editor as *squiggles* under the offending code. The color of the squiggle depends on the severity setting of the rule. The following image shows three violations—one red, one green, and one gray:

Code analyzers inspect code at build time, like legacy analysis if it's enabled, but also live as you type. You can configure the scope of live code analysis to execute for the current document only, all open documents, or the entire solution. See How to: Configure the scope of live code analysis.

> **TIP**
>
> Build-time errors and warnings from code analyzers are shown only if the analyzers are installed as a NuGet package. The built-in analyzers (for example, IDE0067 and IDE0068) never run during build.

Not only do Roslyn code analyzers report the same types of problems that legacy analysis does, but they make it easy for you to fix one or all occurrences of the violation in your file or project. These actions are called *code fixes*. Code fixes are IDE-specific; in Visual Studio, they're implemented as Quick Actions. Not all analyzer diagnostics have an associated code fix.

> **NOTE**
>
> Prior to Visual Studio 2019 16.5 release, **Analyze** > **Run Code Analysis** menu option executes legacy analysis. Starting Visual Studio 2019 16.5, **Run Code Analysis** menu option executes Roslyn-based analyzers for the selected project or solution.

To differentiate between violations from code analyzers and legacy analysis in the Error List, look at the **Tool** column. If the Tool value matches one of the analyzer assemblies in **Solution Explorer**, for example **Microsoft.CodeQuality.Analyzers**, the violation comes from a code analyzer. Otherwise, the violation originates from legacy analysis.

TIP

The **RunCodeAnalysis** MSBuild property in a project file applies only to legacy analysis. If you install analyzers, set **RunCodeAnalysis** to **false** in your project file, to prevent legacy analysis from running after build.

```
<RunCodeAnalysis>false</RunCodeAnalysis>
```

NuGet package versus VSIX extension

Roslyn code analyzers can be installed per-project via a NuGet package. Some are also available as a Visual Studio extension, in which case they apply to any solution you open in Visual Studio. There are some key behavior differences between these two methods of installing analyzers.

Scope

If you install analyzers as a Visual Studio extension, they apply at the solution level and to all instances of Visual Studio. If you install the analyzers as a NuGet package, which is the preferred method, they apply only to the project where the NuGet package was installed. In team environments, analyzers installed as NuGet packages are in scope for *all developers* that work on that project.

Build errors

To have rules enforced at build time, including through the command line or as part of a continuous integration (CI) build, install the analyzers as a NuGet package. Analyzer warnings and errors don't show up in the build report if you install the analyzers as an extension.

The following image shows the command-line build output from building a project that contains an analyzer rule violation:

Rule severity

You cannot configure the severity of rules from analyzers that were installed as a Visual Studio extension. To configure rule severity, install the analyzers as a NuGet package.

Next steps

Install code analyzers in Visual Studio

Use code analyzers in Visual Studio

See also

- Analyzers FAQ
- Write your own code analyzer

- .NET Compiler Platform SDK

Install .NET Compiler Platform code analyzers

8/4/2020 • 2 minutes to read • Edit Online

Visual Studio includes a core set of .NET Compiler Platform (*Roslyn*) analyzers. These analyzers are always on. You can install additional analyzers either as NuGet packages, or as Visual Studio extensions in *VSIX* files.

To install NuGet analyzer packages

1. Find the analyzer package you want to install on www.nuget.org.

   For example, you may want to install the Microsoft FxCop analyzers to check your code for security and performance issues, among others. Or, install StyleCop.Analyzers to look for style issues in your codebase.

2. Install the package in Visual Studio, using either the Package Manager Console or the Package Manager UI.

   > **NOTE**
   >
   > The www.nuget.org page for each analyzer package shows you the command to paste into the **Package Manager Console**. There's even a handy button to copy the text to the clipboard.

   The analyzer assemblies are installed and appear in **Solution Explorer** under **References** > **Analyzers**.

To install VSIX analyzers

1. In Visual Studio, select **Tools** > **Extensions and Updates**.

   The **Extensions and Updates** dialog box opens.

   > **NOTE**
   >
   > Alternatively, you can find and download the analyzer extension directly from Visual Studio Marketplace.

1. In Visual Studio, select **Extensions** > **Manage Extensions**.

   The **Manage Extensions** dialog box opens.

   > **NOTE**
   >
   > Alternatively, you can find and download the analyzer extension directly from Visual Studio Marketplace.

2. Expand **Online** in the left pane, and then select **Visual Studio Marketplace**.

3. In the search box, type the name of the analyzer extension you want to install. For example, you may want to install the Microsoft FxCop analyzers to check your code for security and performance issues, among others.

4. Select **Download**.

   The extension is downloaded.

5. Select **OK** to close the dialog box, and then close all instances of Visual Studio to launch the **VSIX Installer**.

The **VSIX Installer** dialog box opens.

6. Select **Modify** to start the installation.

7. After a minute or two, the installation completes. Select **Close**.

8. Open Visual Studio again.

If you want to check whether the extension is installed, select **Tools** > **Extensions and Updates**. In the **Extensions and Updates** dialog box, select the **Installed** category on the left, and then search for the extension by name.

If you want to check whether the extension is installed, select **Extensions** > **Manage Extensions**. In the **Manage Extensions** dialog box, select the **Installed** category on the left, and then search for the extension by name.

Next steps

Use code analyzers in Visual Studio

See also

- Overview of code analyzers in Visual Studio
- Install FxCop analyzers

Use code analyzers

5/6/2020 • 13 minutes to read • Edit Online

.NET Compiler Platform ("Roslyn") code analyzers analyze your C# or Visual Basic code as you type. Each *diagnostic* or rule has a default severity and suppression state that can be overwritten for your project. This article covers setting rule severity, using rule sets, and suppressing violations.

Analyzers in Solution Explorer

You can do much of the customization of analyzer diagnostics from **Solution Explorer**. If you install analyzers as a NuGet package, an **Analyzers** node appears under the **References** or **Dependencies** node in **Solution Explorer**. If you expand **Analyzers**, and then expand one of the analyzer assemblies, you see all the diagnostics in the assembly.

You can view the properties of a diagnostic, including its description and default severity, in the **Properties** window. To view the properties, right-click on the rule and select **Properties**, or select the rule and then press **Alt+Enter**.

To see online documentation for a diagnostic, right-click on the diagnostic and select **View Help**.

The icons next to each diagnostic in **Solution Explorer** correspond to the icons you see in the rule set when you open it in the editor:

- the "x" in a circle indicates a severity of **Error**
- the "!" in a triangle indicates a severity of **Warning**
- the "i" in a circle indicates a severity of **Info**
- the "i" in a circle on a light-colored background indicates a severity of **Hidden**
- the downward-pointing arrow in a circle indicates that the diagnostic is suppressed

Rule severity

You can configure the severity of analyzer rules, or *diagnostics*, if you install the analyzers as a NuGet package. Starting in Visual Studio 2019 version 16.3, you can configure the severity of a rule in an EditorConfig file. You can also change the severity of a rule from Solution Explorer or in a rule set file.

You can configure the severity of analyzer rules, or *diagnostics*, if you install the analyzers as a NuGet package. You can change the severity of a rule from Solution Explorer or in a rule set file.

The following table shows the different severity options:

SEVERITY (SOLUTION EXPLORER)	SEVERITY (EDITORCONFIG FILE)	BUILD-TIME BEHAVIOR	EDITOR BEHAVIOR
Error	`error`	Violations appear as *Errors* in the Error List and in command-line build output, and cause builds to fail.	Offending code is underlined with a red squiggle and marked by a small red box in the scroll bar.
Warning	`warning`	Violations appear as *Warnings* in the Error List and in command-line build output, but do not cause builds to fail.	Offending code is underlined with a green squiggle and marked by a small green box in the scroll bar.
Info	`suggestion`	Violations appear as *Messages* in the Error List, and not at all in command-line build output.	Offending code is underlined with a gray squiggle and marked by a small gray box in the scroll bar.
Hidden	`silent`	Non-visible to user.	Non-visible to user. The diagnostic is reported to the IDE diagnostic engine, however.
None	`none`	Suppressed completely.	Suppressed completely.
Default	`default`	Corresponds to the default severity of the rule. To determine what the default value for a rule is, look in the Properties window.	Corresponds to the default severity of the rule.

The following screenshot of the code editor shows three different violations with different severities. Notice the color of the squiggle and the small, colored square in the scroll bar on the right.

The following screenshot shows the same three violations as they appear in the Error List:

Set rule severity in an EditorConfig file

(Visual Studio 2019 version 16.3 and later)

You can set the severity for compiler warnings or analyzer rules in an EditorConfig file with the following syntax:

```
dotnet_diagnostic.<rule ID>.severity = <severity>
```

Setting a rule's severity in an EditorConfig file takes precedence over any severity that's set in a rule set or in Solution Explorer. You can manually configure severity in an EditorConfig file or automatically through the light bulb that appears next to a violation.

Set rule severity of multiple analyzer rules at once in an EditorConfig file

(Visual Studio 2019 version 16.5 and later)

You can set the severity for a specific category of analyzer rules or for all analyzer rules with a single entry in an EditorConfig file.

- Set rule severity for a category of analyzer rules:

```
dotnet_analyzer_diagnostic.category-<rule category>.severity = <severity>
```

- Set rule severity for all analyzer rules:

```
dotnet_analyzer_diagnostic.severity = <severity>
```

If you have multiple entries that are applicable to a specific rule ID, the following is the precedence order to choose the applicable entry:

- Severity entry for an individual rule by ID takes precedence over severity entry for a category.
- Severity entry for a category takes precedence over severity entry for all analyzer rules.

Consider the following EditorConfig example, where CA1822 has the category "Performance":

```
[*.cs]
dotnet_diagnostic.CA1822.severity = error
dotnet_analyzer_diagnostic.category-performance.severity = warning
dotnet_analyzer_diagnostic.severity = suggestion
```

In the preceding example, all three entries are applicable to CA1822. However, using the specified precedence rules, the first rule ID-based severity entry wins over the next entries. In this example, CA1822 will have an effective severity of "error". All the remaining rules with the "Performance" category will have severity "warning". All the remaining analyzer rules, which do not have the "Performance" category, will have severity "suggestion".

Manually configure rule severity

1. If you don't already have an EditorConfig file for your project, add one.

2. Add an entry for each rule you want to configure under the corresponding file extension. For example, to set the severity for CA1822 to `error` for C# files, the entry looks as follows:

```
[*.cs]
dotnet_diagnostic.CA1822.severity = error
```

> **NOTE**
>
> For IDE code-style analyzers, you can also configure them in an EditorConfig file using a different syntax, for example, `dotnet_style_qualification_for_field = false:suggestion`. However, if you set a severity using the `dotnet_diagnostic` syntax, it takes precedence. For more information, see Language conventions for EditorConfig.

Convert an existing Ruleset file to EditorConfig file

Starting in Visual Studio 2019 version 16.5, ruleset files are deprecated in favor of the EditorConfig file for analyzer configuration for managed code. Most of the Visual Studio tooling for analyzer rule severity configuration has been updated to work on EditorConfig files instead of ruleset files. EditorConfig files allow you to configure both analyzer rule severities and analyzer options, including Visual Studio IDE code style options. It is highly recommended that you convert your existing ruleset file to an EditorConfig file. It is also recommended that you save the EditorConfig file at the root of your repo or in the solution folder. By using the root of your repo or solution folder, you make sure that the severity settings from this file are automatically applied to the entire repo or solution, respectively.

There are a couple ways to convert an existing ruleset file to an EditorConfig file:

- From the Ruleset Editor in Visual Studio (requires Visual Studio 2019 16.5 or later). If your project already uses a specific ruleset file as its `CodeAnalysisRuleSet`, you can convert it to an equivalent EditorConfig file from Ruleset Editor within Visual Studio.

  1. Double-click the ruleset file in Solution Explorer.

     The Ruleset file should open in the Ruleset Editor. You should see a clickable **infobar** at top of the ruleset editor.

  2. **Click** the infobar link.

     This should open a **Save As** dialog that allows you to select the directory where you want to generate the EditorConfig file.

  3. **Click** the **Save** button to generate the EditorConfig file.

     The generated EditorConfig should open in the editor. Additionally, the MSBuild property `CodeAnalysisRuleSet` gets updated in the project file so that it no longer references the original ruleset file.

- From the command line:

  1. Install the NuGet package Microsoft.CodeAnalysis.RulesetToEditorconfigConverter.

  2. Execute `RulesetToEditorconfigConverter.exe` from the installed package, with paths to ruleset file and EditorConfig file as command-line arguments.

```
Usage: RulesetToEditorconfigConverter.exe <%ruleset_file%> [<%path_to_editorconfig%>]
```

Here is an example ruleset file to convert:

```
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="Rules for ConsoleApp" Description="Code analysis rules for ConsoleApp.csproj."
ToolsVersion="16.0">
  <Rules AnalyzerId="Microsoft.Analyzers.ManagedCodeAnalysis" RuleNamespace="Microsoft.Rules.Managed">
    <Rule Id="CA1001" Action="Warning" />
    <Rule Id="CA1821" Action="Warning" />
    <Rule Id="CA2213" Action="Warning" />
    <Rule Id="CA2231" Action="Warning" />
  </Rules>
</RuleSet>
```

Here is the converted EditorConfig file:

```
# NOTE: Requires **VS2019 16.3** or later

# Rules for ConsoleApp
# Description: Code analysis rules for ConsoleApp.csproj.

# Code files
[*.{cs,vb}]


dotnet_diagnostic.CA1001.severity = warning

dotnet_diagnostic.CA1821.severity = warning

dotnet_diagnostic.CA2213.severity = warning

dotnet_diagnostic.CA2231.severity = warning
```

Automatically configure rule severity

Configure from light bulb menu

Visual Studio provides a convenient way to configure a rule's severity from the Quick Actions light bulb menu.

1. After a violation occurs, hover over the violation squiggle in the editor and open the light bulb menu. Or, put your cursor on the line and press **Ctrl**+. (period).
2. From the light bulb menu, select **Configure or Suppress issues** > **Configure <rule ID> severity**.

3. From there, select one of the severity options.

Visual Studio adds an entry to the EditorConfig file to configure the rule to the requested level, as shown in the preview box.

> **TIP**
>
> If you don't already have an EditorConfig file in the project, Visual Studio creates one for you.

Configure from error list

Visual Studio also provides a convenient way to configure a rule's severity from the error list context menu.

1. After a violation occurs, right-click the diagnostic entry in the error list.
2. From the context menu, select **Set severity**.

3. From there, select one of the severity options.

   Visual Studio adds an entry to the EditorConfig file to configure the rule to the requested level.

> **TIP**
>
> If you don't already have an EditorConfig file in the project, Visual Studio creates one for you.

Set rule severity from Solution Explorer

1. In Solution Explorer, expand **References** > **Analyzers** (or **Dependencies** > **Analyzers** for .NET Core projects).
2. Expand the assembly that contains the rule you want to set severity for.
3. Right-click the rule and select **Set severity**. In the context menu, select one of the severity options.

   Visual Studio adds an entry to the EditorConfig file to configure the rule to the requested level. If your project uses a ruleset file instead of an EditorConfig file, the severity entry is added to the ruleset file.

> **TIP**
> If you don't already have an EditorConfig file or ruleset file in the project, Visual Studio creates a new EditorConfig file for you.

3. Right-click the rule and select **Set Rule Set Severity**. In the context menu, select one of the severity options.

   The severity for the rule is saved in the active rule set file.

Set rule severity in the rule set file

1. Open the active rule set file by double-clicking it in **Solution Explorer**, selecting **Open Active Rule Set** on the right-click menu of the **References** > **Analyzers** node, or by selecting **Open** on the **Code Analysis** property page for the project.

   If this is the first time you're editing the rule set, Visual Studio makes a copy of the default rule set file, names it *<projectname>.ruleset*, and adds it to your project. This custom rule set also becomes the active rule set for your project.

   > **NOTE**
   > .NET Core and .NET Standard projects do not support the menu commands for rule sets in **Solution Explorer**, for example, **Open Active Rule Set**. To specify a non-default rule set for a .NET Core or .NET Standard project, manually add the **CodeAnalysisRuleSet** property to the project file. You can still configure the rules within the rule set in the Visual Studio rule set editor UI.

2. Browse to the rule by expanding its containing assembly.

3. In the **Action** column, select the value to open a drop-down list, and select the desired severity from the list.

Configure Generated code

Analyzers run on all source files in a project and report violations on them. However, these violations are not useful on generated code files, such as designer generated code files, temporary source files generated by build system, etc. Users cannot manually edit these files and/or are not concerned about fixing violations in these kind of tooling-generated files.

By default, the analyzer driver executing analyzers treats files with certain name, file extension, or auto-generated file header as generated code files. For example, a file name ending with `.designer.cs` or `.generated.cs` is considered generated code. However, these heuristics might not be able to identify all the custom generated code files in user's source code.

Starting with Visual Studio 2019 16.5, end users can configure specific files and/or folders to be treated as generated code in an EditorConfig file. Follow the steps below to add such a configuration:

1. If you don't already have an EditorConfig file for your project, add one.
2. Add the `generated_code = true | false` entry for specific files and/or folders. For example, to treat all files whose name ends with `.MyGenerated.cs` as generated code, the entry would be as follows:

   ```
   [*.MyGenerated.cs]
   generated_code = true
   ```

Suppress violations

There are multiple ways to suppress rule violations:

- In an **EditorConfig file**

  Set the severity to `none`, for example, `dotnet_diagnostic.CA1822.severity = none`.

- From the **Analyze** menu

  Select **Analyze** > **Build and Suppress Active Issues** on the menu bar to suppress all current violations. This is sometimes referred to as "baselining".

- From the **Analyze** menu

  Select **Analyze** > **Run Code Analysis and Suppress Active Issues** on the menu bar to suppress all current violations. This is sometimes referred to as "baselining".

- From **Solution Explorer**

  Set the rule's severity to **None**.

- From the **rule set editor**

  Uncheck the box next to its name or set **Action** to **None**.

- From the **code editor**

  Place the cursor in the line of code with the violation and press **Ctrl+Period (.)** to open the **Quick Actions** menu. Select **Suppress CAXXXX** > **in Source/in Suppression File**.

- From the **Error List**

  Select the rules you want to suppress, and then right-click and select **Suppress** > **In Source/In Suppression File**.

  - If you suppress **In Source**, the **Preview Changes** dialog opens and shows a preview of the C# #pragma warning or Visual Basic #Disable warning directive that's added to the source code.

  - If you select **In Suppression File**, the **Preview Changes** dialog opens and shows a preview of the SuppressMessageAttribute attribute that's added to the global suppressions file.

In the **Preview Changes** dialog, select **Apply**.

> **NOTE**
>
> If you don't see the **Suppress** menu option in **Solution Explorer**, the violation is likely coming from build and not live analysis. The **Error List** displays diagnostics, or rule violations, from both live code analysis and build. Since the build diagnostics can be stale, for example, if you've edited the code to fix the violation but haven't rebuilt, you cannot suppress these diagnostics from the **Error List**. Diagnostics from live analysis, or IntelliSense, are always up-to-date with current sources and can be suppressed from the **Error List**. To exclude *build* diagnostics from your selection, switch the **Error List** source filter from **Build + IntelliSense** to **IntelliSense Only**. Then, select the diagnostics you want to suppress and proceed as described previously.
>
>
>

Command-line usage

When you build your project at the command line, rule violations appear in the build output if the following conditions are met:

- The analyzers are installed as a NuGet package and not as a VSIX extension.
- One or more rules are violated in the project's code.
- The severity of a violated rule is set to either **warning**, in which case violations don't cause build to fail, or **error**, in which case violations cause build to fail.

The verbosity of the build output does not affect whether rule violations are shown. Even with **quiet** verbosity, rule violations appear in the build output.

> **TIP**
>
> If you're accustomed to running legacy analysis from the command line, either with *FxCopCmd.exe* or through msbuild with the **RunCodeAnalysis** flag, here's how to do that with code analyzers.

To see analyzer violations at the command line when you build your project using msbuild, run a command like this:

```
msbuild myproject.csproj /target:rebuild /verbosity:minimal
```

The following image shows the command-line build output from building a project that contains an analyzer rule violation:

Dependent projects

In a .NET Core project, if you add a reference to a project that has NuGet analyzers, those analyzers are automatically added to the dependent project too. To disable this behavior, for example if the dependent project is a unit test project, mark the NuGet package as private in the *.csproj* or *.vbproj* file of the referenced project by setting the **PrivateAssets** attribute:

```
<PackageReference Include="Microsoft.CodeAnalysis.FxCopAnalyzers" Version="2.9.0" PrivateAssets="all" />
```

See also

- Overview of code analyzers in Visual Studio
- Submit a code analyzer bug
- Use rule sets
- Suppress code analysis warnings

How to: Configure live code analysis scope for managed code

8/4/2020 • 3 minutes to read • Edit Online

What is "Live code analysis" for managed code?

Visual Studio executes a bunch of live code analyses, also referred to as *background analysis*, while you are editing source files in the editor. Some of it is required minimal analysis for an acceptable Visual Studio IDE editing experience. Some of it is for improved responsiveness for IDE features. While some of it is to enable additional IDE functionality, such as diagnostics and code fixes from Roslyn analyzers. Based on the functionality, these analyses can be grouped as follows:

- **Background computation of diagnostics**: Analysis to compute errors, warnings, and suggestions in source files. These diagnostics show up as entries in the error list and as squiggles in the editor. They can be classified into two categories:
  - C# and Visual Basic compiler diagnostics
  - Roslyn analyzer diagnostics, which includes:
    - Built-in IDE analyzers for code style suggestions and
    - Third party analyzer packages installed for projects in the current solution.
- **Other background analyses**: Analysis to improve the responsiveness and Visual Studio interaction for IDE features. Some examples of such analyses are:
  - Background parsing of open files.
  - Background compilation of projects with open files to realize symbols for improved responsiveness of certain IDE features.
  - Building syntax and symbol caches.
  - Detecting designer association for source files, such as forms, controls, etc.

Default analysis scope

By default, live code analysis for background computation of diagnostics executes for all the files that are *opened* in Visual Studio. Few of the *other background analyses* mentioned above execute for all the projects, which have at least one open file. While few background analyses execute for the entire solution.

Custom analysis scope

The default scope of each background analysis has been tuned for the optimal user experience, functionality, and performance for majority of customer scenarios and solutions. However, there are cases where customers may want to customize this scope to decrease or increase the background analysis. For example:

- Power save mode: If users are running on laptop battery, they may want to minimize the power consumption for longer battery life. In this scenario, they would want to minimize background analysis.
- On-demand code analysis: If users prefer turning off live analyzer execution and manually running code analysis on-demand, they would want to minimize background analysis. See How to: Manually run code analysis on-demand.
- Full solution analysis: If users want to always see all diagnostics in all files in the solution, regardless of whether they are open in the editor or not. In this scenario, they would want to maximize background analysis scope to

entire solution.

Starting in Visual Studio 2019 version 16.5, users can explicitly customize the scope of all live code analysis, including diagnostics computation, for C# and Visual Basic projects. Available analysis scopes are:

- **Current document**: Minimizes the live code analysis scope to only execute for the current or visible file in the editor.
- **Open documents**: Default live code analysis scope, as described in the above section.
- **Entire solution**: Maximizes the live code analysis scope to execute for all files and projects in the entire solution.

You can choose one of the above custom analysis scopes in Tools Options dialog by following the below steps:

1. To open the **Options** dialog box, on the menu bar in Visual Studio choose **Tools** > **Options**.
2. In the **Options** dialog box, choose **Text Editor** > **C#** or **Basic** > **Advanced**.
3. Select the desired **Background analysis scope** to customize the analysis scope. Choose **OK** when you're done.

> **NOTE**
>
> Prior to Visual Studio 2019 version 16.5, users can customize the analysis scope for diagnostics computation to entire solution using the **Enable full solution analysis** check box from **Tools** > **Options** > **Text Editor** > **C#** or **Basic** > **Advanced** tab. There is no support to minimize the background analysis scope in prior Visual Studio versions.

Automatically minimize live code analysis scope

If Visual Studio detects that 200 MB or less of system memory is available to it, it automatically minimizes the live code analysis scope to "Current Document". If this occurs, an alert appears informing you that Visual Studio has disabled some features. A button lets you switch back to the prior analysis scope if you want.

See also

- Automatic feature suspension
- Power save mode feature request

How to: Run code analysis manually for managed code (requires Visual Studio 2019 version 16.5 or later)

8/4/2020 • 2 minutes to read • Edit Online

By default, .NET Compiler Platform ("Roslyn") code analyzers analyze your C# or Visual Basic code as you type by doing live analysis, as well as during build. Hence, you would normally not require to manually trigger code analysis. However, there are some scenarios where you may want to manually trigger code analysis:

- By default, live code analysis executes analyzers only for open files in Visual Studio. However, you may be interested in viewing code analysis warnings for all files in a specific project or solution. If so, you would want to trigger code analysis once on a project or a solution. Alternatively, you can enable continuous live code analysis to execute on entire solution. For more information, see How to: Configure live code analysis scope for managed code.
- You may prefer on-demand code analysis execution workflow over continuous live analysis or build-time analysis. If so, you can disable analyzer execution during live analysis and/or build. For information about disabling analysis, see How to disable source code analysis. Then you would want to manually trigger code analysis once on a project or a solution.

Run code analysis manually

1. In **Solution Explorer**, click the project.
2. On the **Analyze** menu, click **Run Code Analysis on** *Project Name*.

Code analysis will start executing in the background. You should see the message **Running code analysis for <project>...** in the Visual Studio status bar towards the bottom-left corner. Once code analysis completes, the status message will change to **Code analysis completed for <project>**. Error list will soon refresh with all code analysis diagnostics.

How to disable source code analysis for managed code

8/4/2020 • 2 minutes to read • Edit Online

This page helps you disable code analysis in Visual Studio. There are limitations to what you can disable, and the procedure for turning off code analysis differs depending on a few factors:

- Project type (.NET Core/Standard versus .NET Framework)

  .NET Core and .NET Standard projects have options on their Code Analysis properties page that let you turn off code analysis from analyzers installed as a NuGet package. For more information, see .NET Core and .NET Standard projects. To turn off source code analysis for .NET Framework projects, see .NET Framework projects.

- Source analysis versus legacy analysis

  This topic applies to source code analysis and not to legacy (binary) analysis. For information about disabling legacy analysis, see How to: Enable and disable legacy code analysis.

.NET Core and .NET Standard projects

Starting in Visual Studio 2019 version 16.3, there are two checkboxes available in the Code Analysis properties page that let you control whether analyzers run at build time and design time. These options are project-specific.

To open this page, right-click on the project node in **Solution Explorer** and select **Properties**. Select the **Code Analysis** tab.

- To disable source analysis at build time, uncheck the **Run on build** option.

- To disable live source analysis, uncheck the **Run on live analysis** option.

> **NOTE**
> Starting in Visual Studio 2019 version 16.5, if you prefer the on-demand code analysis execution workflow, you can disable analyzer execution during live analysis and/or build and manually trigger code analysis once on a project or a solution on demand. For information about running code analysis manually, see How to: Run Code Analysis Manually for Managed Code.

.NET Framework projects

To turn off source code analysis for analyzers, add one or more of the following MSBuild properties to the project file.

MSBUILD PROPERTY	DESCRIPTION	DEFAULT
`RunAnalyzersDuringBuild`	Controls whether analyzers run at build time.	`true`
`RunAnalyzersDuringLiveAnalysis`	Controls whether analyzers analyze code live at design time.	`true`
`RunAnalyzers`	Disables analyzers at both build and design time. This property takes precedence over `RunAnalyzersDuringBuild` and `RunAnalyzersDuringLiveAnalysis`.	`true`

Examples:

```
<RunAnalyzersDuringBuild>false</RunAnalyzersDuringBuild>
<RunAnalyzersDuringLiveAnalysis>false</RunAnalyzersDuringLiveAnalysis>
<RunAnalyzers>false</RunAnalyzers>
```

Source analysis

You cannot turn off source analysis in Visual Studio 2017. If you want to clear analyzer errors from the Error List, you can suppress all the current violations by choosing **Analyze** > **Run Code Analysis and Suppress Active Issues** on the menu bar. For more information, see Suppress violations.

Starting in Visual Studio 2019 version 16.3, you can turn off source code analysis or execute it on demand. Consider upgrading to Visual Studio 2019.

Legacy analysis

You can disable legacy, build-time analysis on the **Code Analysis** properties page. For more information, see How to: Enable and disable legacy code analysis.

See also

- Suppress violations
- How to: Enable and disable legacy code analysis

Code analysis FAQ

4/21/2020 • 3 minutes to read • Edit Online

This page contains answers to some frequently asked questions about .NET Compiler Platform-based code analysis in Visual Studio.

Code analysis versus EditorConfig

Q: Should I use code analysis or EditorConfig for checking code style?

A: Code analysis and EditorConfig files work hand-in-hand. When you define code styles in an EditorConfig file or on the text editor Options page, you're actually configuring the code analyzers that are built into Visual Studio. EditorConfig files can be used to enable or disable analyzer rules, and also to configure some NuGet analyzer packages, such as FxCop analyzers.

EditorConfig versus rule sets

Q: Should I configure my analyzers using a rule set or an EditorConfig file?

A: Rule sets and EditorConfig files can coexist and can both be used to configure analyzers. Both EditorConfig files and rule sets let you enable and disable rules and set their severity.

However, EditorConfig files offer additional ways to configure rules too:

- For FxCop analyzers, EditorConfig files let you define which types of code to analyze.
- For the code-style analyzers that are built into Visual Studio, EditorConfig files let you define the preferred code styles for a codebase.

In addition to rule sets and EditorConfig files, some analyzers are configured through the use of text files marked as additional files for the C# and VB compilers.

> **NOTE**
>
> - EditorConfig files can only be used to enable rules and set their severity in Visual Studio 2019 version 16.3 and later.
> - EditorConfig files cannot be used to configure legacy analysis, whereas rule sets can.

Code analysis in CI builds

Q: Does .NET Compiler Platform-based code analysis work in continuous integration (CI) builds?

A: Yes. For analyzers that are installed from a NuGet package, those rules are enforced at build time, including during a CI build. Analyzers used in CI builds respect rule configuration from both rule sets and EditorConfig files. Currently, the code analyzers that are built into Visual Studio are not available as a NuGet package, and so these rules are not enforceable in a CI build.

IDE analyzers versus StyleCop

Q: What's the difference between the Visual Studio IDE code analyzers and StyleCop analyzers?

A: The Visual Studio IDE includes built-in analyzers that look for both code style and quality issues. These rules help you use new language features as they're introduced and improve the maintainability of your code. IDE analyzers are continually updated with each Visual Studio release.

StyleCop analyzers are third-party analyzers installed as a NuGet package that check for style consistency in your code. In general, StyleCop rules let you set personal preferences for a code base without recommending one style over another.

Code analyzers versus legacy analysis

Q: What's the difference between legacy analysis and .NET Compiler Platform-based code analysis?

A: .NET Compiler Platform-based code analysis analyzes source code in real time and during compilation, whereas legacy analysis analyzes binary files after build has completed. For more information, see .NET Compiler Platform-based analysis versus legacy analysis and FxCop analyzers FAQ.

Treat warnings as errors

Q: My project uses the build option to treat warnings as errors. After migrating from legacy analysis to source code analysis, all of the code analysis warnings are now appearing as errors. How can I prevent that?

A: To prevent code analysis warnings from being treated as errors, follow these steps:

1. Create a .props file with the following content:

```
<Project>
   <PropertyGroup>
      <CodeAnalysisTreatWarningsAsErrors>false</CodeAnalysisTreatWarningsAsErrors>
   </PropertyGroup>
</Project>
```

2. Add a line to your .csproj or .vbproj project file to import the .props file you created in the previous step. This line must be placed before any lines that import the FxCop analyzer .props files. For example, if your .props file is named codeanalysis.props:

```
...
<Import Project="..\..\codeanalysis.props" Condition="Exists('..\..\codeanalysis.props')" />
<Import
Project="..\packages\Microsoft.CodeAnalysis.FxCopAnalyzers.2.6.5\build\Microsoft.CodeAnalysis.FxCopAnaly
zers.props"
Condition="Exists('..\packages\Microsoft.CodeAnalysis.FxCopAnalyzers.2.6.5\build\Microsoft.CodeAnalysis.
FxCopAnalyzers.props')" />
...
```

Code analysis solution property page

Q: Where is the Code Analysis property page for the solution?

A: The Code Analysis property page at the solution level was removed in favor of the more reliable shared property group. For managing Code Analysis at the project level, the Code Analysis property page is still available. (For managed projects, we also recommend migrating from rulesets to EditorConfig for rule configuration.) For sharing rulesets across multiple/all projects in a solution or a repo, we recommend defining a property group with CodeAnalysisRuleSet property in a shared props/targets file or Directory.props/Directory.targets file. If you don't have any such common props or targets that all your projects import, you should consider adding such a property group to a Directory.props or a Directory.targets at a top level solution directory, which is automatically imported in all project files defined in the directory or its sub-directories.

See also

- Analyzers overview
- .NET coding convention settings for EditorConfig

Install FxCop analyzers in Visual Studio

8/4/2020 • 3 minutes to read • Edit Online

Microsoft created a set of analyzers, called Microsoft.CodeAnalysis.FxCopAnalyzers, that contains the most important "FxCop" rules from legacy analysis. These analyzers check your code for security, performance, and design issues, among others.

You can install these FxCop analyzers either as a NuGet package or as a VSIX extension to Visual Studio. To learn about the pros and cons of each, see NuGet package vs. VSIX extension.

NuGet package

In Visual Studio 2019 version 16.3 and later, you can install the Microsoft.CodeAnalysis.FxCopAnalyzers NuGet package directly from the project's Code Analysis properties page:

1. Right-click on the project node in **Solution Explorer**, select **Properties**, and then select the **Code Analysis** tab.

2. Select **Install**.

   Visual Studio installs the latest version of the Microsoft.CodeAnalysis.FxCopAnalyzers package. The assemblies appear in **Solution Explorer** under **References** > **Analyzers**.

If you're using an older version of Visual Studio 2019, install the package using either the Package Manager Console or the Package Manager UI.

1. Determine which analyzer package version to install, based on your version of Visual Studio.
2. Install the package in Visual Studio using either the Package Manager Console or the Package Manager UI.

> **NOTE**
>
> The nuget.org page for each analyzer package shows you the command to paste into the **Package Manager Console**. There's even a handy button to copy the text to the clipbboard.
>
>
>

The analyzer assemblies are installed, and they appear in **Solution Explorer** under **References** > **Analyzers**.

Custom installation

For custom installation, for example to specify a different version of the package, select the ellipsis (...) button on the project's Code Analysis properties page. This button opens the NuGet package manager with "Microsoft.CodeAnalysis.FxCopAnalyzers" as the search string.

> **TIP**
> Determine which analyzer package version to install, based on your version of Visual Studio. You can also install the package from the Package Manager UI.

FxCopAnalyzers package versions

Use the following guidelines to determine which version of the FxCop analyzers package to install for your version of Visual Studio:

VISUAL STUDIO VERSION	FXCOP ANALYZER PACKAGE VERSION
Visual Studio 2019 (all versions)	latest
Visual Studio 2017 version 15.9	2.9.9
Visual Studio 2017 version 15.5 to 15.8	2.6.4
Visual Studio 2017 version 15.3 to 15.4	2.3.0-beta1
Visual Studio 2017 version 15.0 to 15.2	2.0.0-beta2
Visual Studio 2015 update 2 and 3	1.2.0-beta2
Visual Studio 2015 Update 1	1.1.0
Visual Studio 2015 RTW	1.0.1

VSIX

On Visual Studio 2017 version 15.5 and later, you can install the Microsoft Code Analysis 2017 extension that contains all of the FxCop analyzers for managed projects.

1. In Visual Studio, select **Tools** > **Extensions and Updates**.

The **Extensions and Updates** dialog box opens.

> **NOTE**
>
> Alternatively, download the extension directly from Visual Studio Marketplace.

2. Expand **Online** in the left pane, and then select **Visual Studio Marketplace**.

3. In the search box, type "code analysis", and look for the **Microsoft Code Analysis 2017** extension.

The Microsoft Code Analysis 2019 extension contains all of the FxCop analyzers for managed projects. To install this extension:

1. In Visual Studio, select **Extensions** > **Manage Extensions**.

   The **Manage Extensions** dialog box opens.

   > **NOTE**
   >
   > Alternatively, download the extension directly from Visual Studio Marketplace.

2. Expand **Online** in the left pane, and then select **Visual Studio Marketplace**.

3. In the search box, type "code analysis", and look for the **Microsoft Code Analysis 2019** extension.

4. Select **Download**.

The extension is downloaded.

5. Select **OK** to close the dialog box, and then close all instances of Visual Studio to launch the **VSIX Installer**.

   The **VSIX Installer** dialog box opens.

6. Select **Modify** to start the installation.

   After a minute or two, the installation completes.

7. Select **Close**, then open Visual Studio again.

If you want to check whether the extension is installed, select **Tools** > **Extensions and Updates**. In the **Extensions and Updates** dialog box, select the **Installed** category on the left, and then search for the extension by name.

If you want to check whether the extension is installed, select **Extensions** > **Manage Extensions**. In the **Manage Extensions** dialog box, select the **Installed** category on the left, and then search for the extension by name.

See also

- Overview of code analyzers in Visual Studio
- Use code analyzers in Visual Studio
- Migrate from legacy analysis to code analyzers

Migrate from legacy analysis (FxCop) to source analysis (FxCop analyzers)

3/7/2020 • 2 minutes to read • Edit Online

Source analysis by .NET Compiler Platform ("Roslyn") analyzers replaces legacy analysis for managed code. For newer project templates such as .NET Core and .NET Standard projects, legacy analysis is not available.

Many of the legacy analysis (FxCop) rules have already been rewritten for FxCop analyzers, a set of Roslyn code analyzers. You install FxCop analyzers as a NuGet package that's referenced by the project or solution. FxCop analyzers run source-code based analysis during compiler execution. Analyzer results are reported along with compiler results.

For more information on the differences between legacy analysis and source analysis, see the following:

- Source code analysis versus legacy analysis
- FAQ about FxCop analyzers

To migrate to source analysis, install the FxCop analyzers. Like legacy analysis rule violations, source code analysis violations appear in the Error List window in Visual Studio. In addition, source code analysis violations also show up in the code editor as *squiggles* under the offending code. The color of the squiggle depends on the severity setting of the rule. To see the status of rules ported to the new FxCop analyzers, see Ported and unported rules.

To learn more about how to configure the FxCop analyzers:

- To configure FxCop analyzers, see Configure FxCop analyzers.
- To find out about configuring analyzers using predefined rules with EditorConfig or a rule set file, see Enable a category of rules.

Frequently asked questions about FxCop and FxCop analyzers

3/16/2020 • 2 minutes to read • Edit Online

It can be a little confusing to understand the differences between legacy FxCop and FxCop analyzers. This article aims to address some of questions you might have.

What's the difference between legacy FxCop and FxCop analyzers?

Legacy FxCop runs post-build analysis on a compiled assembly. It runs as a separate executable called **FxCopCmd.exe**. FxCopCmd.exe loads the compiled assembly, runs code analysis, and then reports the results (or *diagnostics*).

FxCop analyzers are based on the .NET Compiler Platform ("Roslyn"). You install them as a NuGet package that's referenced by the project or solution. FxCop analyzers run source-code based analysis during compiler execution. FxCop analyzers are hosted within the compiler process, either **csc.exe** or **vbc.exe**, and run analysis when the project is built. Analyzer results are reported along with compiler results.

> **NOTE**
>
> You can also install FxCop analyzers as a Visual Studio extension. In this case, the analyzers execute as you type in the code editor, but they don't execute at build time. If you want to run FxCop analyzers as part of continuous integration (CI), install them as a NuGet package instead.

Does the Run Code Analysis command run FxCop analyzers?

Prior to Visual Studio 2019 16.5 release, when you select **Analyze** > **Run Code Analysis**, it executes legacy analysis. Starting Visual Studio 2019 16.5, **Run Code Analysis** menu option executes Roslyn-based analyzers for the selected project or solution. If you have installed Roslyn-based FxCop analyzers, they would also be executed. For more information, see How to: Run Code Analysis Manually for Managed Code.

Does the RunCodeAnalysis msbuild project property run analyzers?

No. The **RunCodeAnalysis** property in a project file (for example, *.csproj*) is only used to execute legacy FxCop. It runs a post-build msbuild task that invokes **FxCopCmd.exe**.

So how do I run FxCop analyzers then?

To run FxCop analyzers, first install the NuGet package for them. Then build your project or solution from Visual Studio or using msbuild. The warnings and errors that the FxCop analyzers generate will appear in the **Error List** or the command window.

I get warning CA0507 even after I've installed the FxCop analyzers NuGet package

If you've installed FxCop analyzers but continue to get warning CA0507 **""Run Code Analysis" has been deprecated in favor of FxCop analyzers, which run during build"**, you may need to set the **RunCodeAnalysis** msbuild property in your project file to **false**. Otherwise, legacy analysis will execute after

each build.

```
<RunCodeAnalysis>false</RunCodeAnalysis>
```

Which rules have been ported to FxCop analyzers?

For information about which legacy analysis rules have been ported to FxCop analyzers, see Fxcop rule port status.

Code analysis warnings are treated as errors

If your project uses the build option to treat warnings as errors, FxCop analyzer warnings may appear as errors. To prevent code analysis warnings from being treated as errors, follow the steps at Code analysis FAQ.

See also

- Overview of .NET Compiler Platform analyzers
- Migrate to FxCop analyzers
- Install FxCop analyzers

Fxcop rule port status

4/28/2020 • 8 minutes to read • Edit Online

If you previously used static code analysis in Visual Studio, you may be wondering which of those rules are available in the current implementation as FxCop analyzers. This page lists the rules that are ported as well as those that haven't been ported and whether there are plans to port them.

Ported rules

The autogenerated documentation page in the roslyn-analyzers repo has the most up-to-date list of rules that have been ported to FxCop analyzers. That page also has additional information, such as whether the rule is enabled by default and if it has an associated *code fix*. (Code fixes are one-click fixes available in the light bulb icon menu in Visual Studio.)

As of the date on this page, the list of FxCop rules that have been ported to FxCop analyzers includes:

RULE ID	TITLE
CA1000	Do not declare static members on generic types
CA1001	Types that own disposable fields should be disposable
CA1003	Use generic event handler instances
CA1008	Enums should have zero value
CA1010	Collections should implement generic interface
CA1012	Abstract types should not have constructors
CA1014	Mark assemblies with CLSCompliant
CA1016	Mark assemblies with assembly version
CA1017	Mark assemblies with ComVisible
CA1018	Mark attributes with AttributeUsageAttribute
CA1019	Define accessors for attribute arguments
CA1021	Avoid out parameters
CA1024	Use properties where appropriate
CA1027	Mark enums with FlagsAttribute
CA1028	Enum Storage should be Int32
CA1030	Use events where appropriate

RULE ID	TITLE
CA1031	Do not catch general exception types
CA1032	Implement standard exception constructors
CA1033	Interface methods should be callable by child types
CA1034	Nested types should not be visible
CA1036	Override methods on comparable types
CA1040	Avoid empty interfaces
CA1041	Provide ObsoleteAttribute message
CA1043	Use Integral Or String Argument For Indexers
CA1044	Properties should not be write only
CA1050	Declare types in namespaces
CA1051	Do not declare visible instance fields
CA1052	Static holder types should be static or NotInheritable
CA1053	Static holder types should not have constructors (CA1053 is part of CA1052 for FxCop analyzers)
CA1054	Uri parameters should not be strings
CA1055	Uri return values should not be strings
CA1056	Uri properties should not be strings
CA1058	Types should not extend certain base types
CA1060	Move pinvokes to native methods class
CA1061	Do not hide base class methods
CA1062	Validate arguments of public methods
CA1063	Implement IDisposable Correctly
CA1064	Exceptions should be public
CA1065	Do not raise exceptions in unexpected locations
CA1066	Type {0} should implement IEquatable<T> because it overrides Equals

RULE ID	TITLE
CA1067	Override Object.Equals(object) when implementing IEquatable<T>
CA1068	CancellationToken parameters must come last
CA1200	Avoid using cref tags with a prefix
CA1303	Do not pass literals as localized parameters
CA1304	Specify CultureInfo
CA1305	Specify IFormatProvider
CA1307	Specify StringComparison
CA1308	Normalize strings to uppercase
CA1309	Use ordinal string comparison
CA1401	P/Invokes should not be visible
CA1501	Avoid excessive inheritance
CA1502	Avoid excessive complexity
CA1505	Avoid unmaintainable code
CA1506	Avoid excessive class coupling
CA1507	Use nameof to express symbol names
CA1508	Avoid dead conditional code
CA1509	Invalid entry in code metrics rule specification file
CA1707	Identifiers should not contain underscores
CA1708	Identifiers should differ by more than case
CA1710	Identifiers should have correct suffix
CA1711	Identifiers should not have incorrect suffix
CA1712	Do not prefix enum values with type name
CA1714	Flags enums should have plural names
CA1715	Identifiers should have correct prefix
CA1716	Identifiers should not match keywords

RULE ID	TITLE
CA1717	Only FlagsAttribute enums should have plural names
CA1720	Identifier contains type name
CA1721	Property names should not match get methods
CA1724	Type names should not match namespaces
CA1725	Parameter names should match base declaration
CA1801	Review unused parameters
CA1802	Use literals where appropriate
CA1806	Do not ignore method results
CA1810	Initialize reference type static fields inline
CA1812	Avoid uninstantiated internal classes
CA1813	Avoid unsealed attributes
CA1814	Prefer jagged arrays over multidimensional
CA1815	Override equals and operator equals on value types
CA1816	Dispose methods should call SuppressFinalize
CA1819	Properties should not return arrays
CA1820	Test for empty strings using string length
CA1821	Remove empty Finalizers
CA1822	Mark members as static
CA1823	Avoid unused private fields
CA1824	Mark assemblies with NeutralResourcesLanguageAttribute
CA1825	Avoid zero-length array allocations.
CA1826	Do not use Enumerable methods on indexable collections. Instead use the collection directly
CA2000	Dispose objects before losing scope
CA2002	Do not lock on objects with weak identity
CA2007	Consider calling ConfigureAwait on the awaited task

RULE ID	TITLE
CA2008	Do not create tasks without passing a TaskScheduler
CA2009	Do not call ToImmutableCollection on an ImmutableCollection value
CA2010	Always consume the value returned by methods marked with PreserveSigAttribute
CA2100	Review SQL queries for security vulnerabilities
CA2101	Specify marshaling for P/Invoke string arguments
CA2119	Seal methods that satisfy private interfaces
CA2153	Do Not Catch Corrupted State Exceptions
CA2200	Rethrow to preserve stack details.
CA2201	Do not raise reserved exception types
CA2207	Initialize value type static fields inline
CA2208	Instantiate argument exceptions correctly
CA2211	Non-constant fields should not be visible
CA2213	Disposable fields should be disposed
CA2214	Do not call overridable methods in constructors
CA2216	Disposable types should declare finalizer
CA2217	Do not mark enums with FlagsAttribute
CA2218	Override GetHashCode on overriding Equals
CA2219	Do not raise exceptions in finally clauses
CA2224	Override Equals on overloading operator equals
CA2225	Operator overloads have named alternates
CA2226	Operators should have symmetrical overloads
CA2227	Collection properties should be read only
CA2229	Implement serialization constructors
CA2231	Overload operator equals on overriding value type Equals

RULE ID	TITLE
CA2234	Pass system uri objects instead of strings
CA2235	Mark all non-serializable fields
CA2237	Mark ISerializable types with serializable
CA2241	Provide correct arguments to formatting methods
CA2242	Test for NaN correctly
CA2243	Attribute string literals should parse correctly
CA2244	Do not duplicate indexed element initializations
CA2300	Do not use insecure deserializer BinaryFormatter
CA2301	Do not call BinaryFormatter.Deserialize without first setting BinaryFormatter.Binder
CA2302	Ensure BinaryFormatter.Binder is set before calling BinaryFormatter.Deserialize
CA2305	Do not use insecure deserializer LosFormatter
CA2310	Do not use insecure deserializer NetDataContractSerializer
CA2311	Do not deserialize without first setting NetDataContractSerializer.Binder
CA2312	Ensure NetDataContractSerializer.Binder is set before deserializing
CA2315	Do not use insecure deserializer ObjectStateFormatter
CA2321	Do not deserialize with JavaScriptSerializer using a SimpleTypeResolver
CA2322	Ensure JavaScriptSerializer is not initialized with SimpleTypeResolver before deserializing
CA3001	Review code for SQL injection vulnerabilities
CA3002	Review code for XSS vulnerabilities
CA3003	Review code for file path injection vulnerabilities
CA3004	Review code for information disclosure vulnerabilities
CA3005	Review code for LDAP injection vulnerabilities
CA3006	Review code for process command injection vulnerabilities

RULE ID	TITLE
CA3007	Review code for open redirect vulnerabilities
CA3008	Review code for XPath injection vulnerabilities
CA3009	Review code for XML injection vulnerabilities
CA3010	Review code for XAML injection vulnerabilities
CA3011	Review code for DLL injection vulnerabilities
CA3012	Review code for regex injection vulnerabilities
CA3061	Do Not Add Schema By URL
CA3075	Insecure DTD processing in XML
CA3076	Insecure XSLT script processing.
CA3077	Insecure Processing in API Design, XmlDocument, and XmlTextReader
CA3147	Mark Verb Handlers With Validate Antiforgery Token
CA5350	Do Not Use Weak Cryptographic Algorithms
CA5351	Do Not Use Broken Cryptographic Algorithms
CA5358	Do Not Use Unsafe Cipher Modes
CA5359	Do Not Disable Certificate Validation
CA5360	Do Not Call Dangerous Methods In Deserialization
CA5361	Do Not Disable SChannel Use of Strong Crypto
CA5362	Do Not Refer Self In Serializable Class
CA5363	Do Not Disable Request Validation
CA5364	Do Not Use Deprecated Security Protocols
CA5365	Do Not Disable HTTP Header Checking
CA5366	Use XmlReader For DataSet Read Xml
CA5367	Do Not Serialize Types With Pointer Fields
CA5368	Set ViewStateUserKey For Classes Derived From Page
CA5369	Use XmlReader For Deserialize

RULE ID	TITLE
CA5370	Use XmlReader For Validating Reader
CA5371	Use XmlReader For Schema Read
CA5372	Use XmlReader For XPathDocument
CA5373	Do not use obsolete key derivation function
CA5374	Do Not Use XslTransform
CA5375	Do Not Use Account Shared Access Signature
CA5376	Use SharedAccessProtocol HttpsOnly
CA5377	Use Container Level Access Policy
CA5378	Do not disable ServicePointManagerSecurityProtocols
CA5379	Do Not Use Weak Key Derivation Function Algorithm
CA9999	Analyzer version mismatch

Unported rules

The set of rules that hasn't been ported to FxCop analyzers consists of rules that haven't yet but still may be ported, and those that are deprecated and won't be ported.

Rules that may be ported

The following FxCop legacy analysis rules haven't yet been implemented as analyzers, but still may be. This could be because of a blocking technical reason or simply that the rule is lower priority. For more information about the porting status of each rule, click the link in the **Tracking issue** column.

RULE ID	TRACKING ISSUE
CA1002	https://github.com/dotnet/roslyn-analyzers/issues/369
CA1004	https://github.com/dotnet/roslyn-analyzers/issues/370
CA1005	https://github.com/dotnet/roslyn-analyzers/issues/371
CA1006	https://github.com/dotnet/roslyn-analyzers/issues/372
CA1007	https://github.com/dotnet/roslyn-analyzers/issues/373
CA1011	https://github.com/dotnet/roslyn-analyzers/issues/375
CA1021	https://github.com/dotnet/roslyn-analyzers/issues/377
CA1023	https://github.com/dotnet/roslyn-analyzers/issues/378

RULE ID	TRACKING ISSUE
CA1045	https://github.com/dotnet/roslyn-analyzers/issues/391
CA1046	https://github.com/dotnet/roslyn-analyzers/issues/392
CA1047	https://github.com/dotnet/roslyn-analyzers/issues/393
CA1048	https://github.com/dotnet/roslyn-analyzers/issues/394
CA1049	https://github.com/dotnet/roslyn-analyzers/issues/395
CA1057	https://github.com/dotnet/roslyn-analyzers/issues/401
CA1300	https://github.com/dotnet/roslyn-analyzers/issues/408
CA1301	https://github.com/dotnet/roslyn-analyzers/issues/409
CA1306	https://github.com/dotnet/roslyn-analyzers/issues/414
CA1402	https://github.com/dotnet/roslyn-analyzers/issues/418
CA1403	https://github.com/dotnet/roslyn-analyzers/issues/419
CA1404	https://github.com/dotnet/roslyn-analyzers/issues/420
CA1405	https://github.com/dotnet/roslyn-analyzers/issues/421
CA1407	https://github.com/dotnet/roslyn-analyzers/issues/423
CA1408	https://github.com/dotnet/roslyn-analyzers/issues/424
CA1409	https://github.com/dotnet/roslyn-analyzers/issues/425
CA1410	https://github.com/dotnet/roslyn-analyzers/issues/426
CA1411	https://github.com/dotnet/roslyn-analyzers/issues/427
CA1412	https://github.com/dotnet/roslyn-analyzers/issues/428
CA1413	https://github.com/dotnet/roslyn-analyzers/issues/429
CA1414	https://github.com/dotnet/roslyn-analyzers/issues/430
CA1415	https://github.com/dotnet/roslyn-analyzers/issues/431
CA1500	https://github.com/dotnet/roslyn-analyzers/issues/432
CA1600	https://github.com/dotnet/roslyn-analyzers/issues/438
CA1601	https://github.com/dotnet/roslyn-analyzers/issues/439

RULE ID	TRACKING ISSUE
CA1700	https://github.com/dotnet/roslyn-analyzers/issues/440
CA1704	https://github.com/dotnet/roslyn-analyzers/issues/443
CA1709	https://github.com/dotnet/roslyn-analyzers/issues/445
CA1713	https://github.com/dotnet/roslyn-analyzers/issues/449
CA1719	https://github.com/dotnet/roslyn-analyzers/issues/453
CA1722	https://github.com/dotnet/roslyn-analyzers/issues/455
CA1726	https://github.com/dotnet/roslyn-analyzers/issues/458
CA1804	https://github.com/dotnet/roslyn-analyzers/issues/461
CA1811	https://github.com/dotnet/roslyn-analyzers/issues/464
CA1900	https://github.com/dotnet/roslyn-analyzers/issues/474
CA2001	https://github.com/dotnet/roslyn-analyzers/issues/477
CA2004	https://github.com/dotnet/roslyn-analyzers/issues/479
CA2006	https://github.com/dotnet/roslyn-analyzers/issues/480
CA2109	https://github.com/dotnet/roslyn-analyzers/issues/488
CA2204	https://github.com/dotnet/roslyn-analyzers/issues/529
CA2205	https://github.com/dotnet/roslyn-analyzers/issues/530
CA2212	https://github.com/dotnet/roslyn-analyzers/issues/534
CA2215	https://github.com/dotnet/roslyn-analyzers/issues/535
CA2232	https://github.com/dotnet/roslyn-analyzers/issues/545
CA2236	https://github.com/dotnet/roslyn-analyzers/issues/548
CA2238	https://github.com/dotnet/roslyn-analyzers/issues/549
CA2239	https://github.com/dotnet/roslyn-analyzers/issues/550
CA2240	https://github.com/dotnet/roslyn-analyzers/issues/551

Deprecated rules

The following FxCop legacy analysis rules are deprecated and won't be implemented as analyzers. For further information, you can search by rule ID (for example, **CA1009**) on the roslyn-analyzers GitHub issues page.

- CA1009

- CA1020
- CA1025
- CA1026
- CA1035
- CA1038
- CA1039
- CA1059
- CA1302
- CA1400
- CA1406
- CA1504
- CA1701
- CA1702
- CA1703
- CA1800
- CA1809
- CA1901
- CA1903
- CA2003
- CA2102
- CA2103
- CA2104
- CA2105
- CA2106
- CA2107
- CA2108
- CA2111
- CA2112
- CA2114
- CA2115
- CA2116
- CA2117
- CA2118
- CA2120
- CA2121
- CA2122
- CA2123
- CA2124
- CA2126
- CA2130
- CA2131
- CA2132
- CA2133
- CA2134
- CA2135
- CA2136

- CA2137
- CA2138
- CA2139
- CA2140
- CA2141
- CA2142
- CA2143
- CA2144
- CA2145
- CA2146
- CA2147
- CA2149
- CA2151
- CA2202
- CA2210
- CA2220
- CA2221
- CA2222 (justification)
- CA2223
- CA2228
- CA2230
- CA2233
- CA5122

See also

- Microsoft.CodeAnalysis.FxCopAnalyzers rules

Configure FxCop analyzers

5/6/2020 • 2 minutes to read • Edit Online

The FxCop analyzers package consists of the most important "FxCop" rules from legacy analysis converted to .NET Compiler Platform-based code analyzers. For certain FxCop rules, you can refine which parts of your codebase they should be applied to through configurable options. Each option is specified by adding a key-value pair to an EditorConfig file. A configuration file can be specific to a project or it can be shared between two or more projects.

> **TIP**
>
> Add an .editorconfig file to your project by right-clicking on the project in **Solution Explorer** and selecting **Add** > **New Item**. In the **Add New Item** window, enter **editorconfig** in the search box. Select the **editorconfig File (default)** template and choose **Add**.

For information about configuring a rule's severity (for example, whether it's an error or a warning), see Set rule severity in an EditorConfig file. Or, you can choose one of the built-in EditorConfig files or rule sets to quickly enable or disable a category of rules.

The remainder of this article discusses general syntax for the options that refine where FxCop rules are applied.

> **NOTE**
>
> You cannot configure legacy FxCop rules by using an EditorConfig file. For information about the differences between legacy analysis and FxCop analyzers, see FxCop analyzers FAQ.

Option scopes

Each refining option can be configured for all rules, for a category of rules (for example, Naming or Design), or for a specific rule.

All rules

The syntax for configuring an option for *all* rules is as follows:

SYNTAX	EXAMPLE
dotnet_code_quality.OptionName = OptionValue	`dotnet_code_quality.api_surface = public`

Category of rules

The syntax for configuring an option for a *category* of rules (such as Naming, Design, or Performance) is as follows:

SYNTAX	EXAMPLE
dotnet_code_quality.RuleCategory.OptionName = OptionValue	`dotnet_code_quality.Naming.api_surface = public`

Specific rule

The syntax for configuring an option for a *specific* rule is as follows:

SYNTAX	EXAMPLE
dotnet_code_quality.RuleId.OptionName = OptionValue	`dotnet_code_quality.CA1040.api_surface = public`

Enabling Editorconfig based configuration

VS2019 16.3 and later + FxCopAnalyzers package version 3.3.x and later

EditorConfig-based analyzer configuration can be enabled for the following scopes:

- Specific document(s)
- Specific folder(s)
- Specific project(s)
- Specific solution(s)
- Entire repo

To enable configuration, add an *.editorconfig* file with the options in the corresponding directory. This file can also contain EditorConfig-based diagnostic severity configuration entries. See here for more details.

Prior to VS2019 16.3 or using an FxCopAnalyzers package version prior to 3.3.x

Per-project configuration

To enable EditorConfig-based analyzer configuration for a specific project, add an *.editorconfig* file to the project's root directory.

Shared configuration

You can share an .editorconfig file for FxCop analyzer configuration between two or more projects, but it requires some additional steps.

1. Save the *.editorconfig* file to a common location.

2. Create a *.props* file with the following content:

```
<Project DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <SkipDefaultEditorConfigAsAdditionalFile>true</SkipDefaultEditorConfigAsAdditionalFile>
  </PropertyGroup>
  <ItemGroup Condition="Exists('<your path>\.editorconfig')" >
    <AdditionalFiles Include="<your path>\.editorconfig" />
  </ItemGroup>
</Project>
```

3. Add a line to your *.csproj* or *.vbproj* file to import the *.props* file you created in the previous step. This line must be placed before any lines that import the FxCop analyzer *.props* files. For example, if your .props file is named *editorconfig.props*:

```
...
<Import Project="..\..\editorconfig.props" Condition="Exists('..\..\editorconfig.props')"
/>
<Import
Project="..\packages\Microsoft.CodeAnalysis.FxCopAnalyzers.2.6.3\build\Microsoft.CodeAnalys
is.FxCopAnalyzers.props"
Condition="Exists('..\packages\Microsoft.CodeAnalysis.FxCopAnalyzers.2.6.3\build\Microsoft.
CodeAnalysis.FxCopAnalyzers.props')" />
...
```

4. Reload the project.

> **NOTE**
>
> The arbitrary shared location of the EditorConfig file described here applies only to configuring the scope of certain FxCop analyzer rules. For other settings, such as rule severity, general editor settings, and code style, the EditorConfig file must always be placed in the project folder or a parent folder.

See also

- Rule scope options for FxCop analyzers
- Analyzer configuration
- FxCop analyzers
- .NET coding conventions for EditorConfig

Rule scope options for FxCop analyzers

1/1/2020 • 4 minutes to read • Edit Online

Some FxCop analyzer rules let you refine which parts of your codebase they should be applied to. This page lists the available scope configuration options, their allowable values, and the rules to which they can be applied. To use these options, specify them in an EditorConfig file.

These configuration options are available starting in version 2.6.3 of the Microsoft.CodeAnalysis.FxCopAnalyzers NuGet package.

> **TIP**
>
> To see the full list of options that are available for a given version of the FxCopAnalyzers package, look at the *Analyzer Configuration.md* file in the *documentation* folder for the package. The file is located at *%USERPROFILE%\.nuget\packages\microsoft.codeanalysis.fxcopanalyzers\<version>\documentation\Analyzer Configuration.md*. This configuration documentation file is included with each version of the package, starting in version 2.6.5. Here's an example of how an option is documented in the *Analyzer Configuration.md* file:
>
> Option Name: `sufficient_IterationCount_for_weak_KDF_algorithm`
> Option Values: integral values
> Default Value: Specific to each configurable rule ('100000' by default for most rules)
> Example: `dotnet_code_quality.CA5387.sufficient_IterationCount_for_weak_KDF_algorithm = 100000`

api_surface

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES

Which part of the API surface to analyze	`public`
	`internal` or `friend`
	`private`
	`all`
	
	Separate multiple values with a comma (,)
		`public`	CA1000 CA1003 CA1008 CA1010
			CA1012 CA1024 CA1027 CA1028
			CA1030 CA1036 CA1040 CA1041
			CA1043 CA1044 CA1051 CA1052
			CA1054 CA1055 CA1056 CA1058
			CA1063 CA1708 CA1710 CA1711
			CA1714 CA1715 CA1716 CA1717
			CA1720 CA1721 CA1725 CA1801
			CA1802 CA1815 CA1819 CA2217
			CA2225 CA2226 CA2231 CA2234

exclude_async_void_methods

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Whether to ignore asynchronous methods that don't return a value	`true` `false`	`false`	CA2007

> **NOTE**
> In version 2.6.3 and earlier of the analyzer package, this option was named `skip_async_void_methods`.

exclude_single_letter_type_parameters

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Whether to exclude single-character type parameters from the rule, for example, `S` in `Collection<S>`	`true` `false`	`false`	CA1715

> **NOTE**
> In version 2.6.3 and earlier of the analyzer package, this option was named `allow_single_letter_type_parameters`.

output_kind

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Specifies that code in a project that generates this type of assembly should be analyzed	One or more fields of the OutputKind enumeration Separate multiple values with a comma (,)	All output kinds	CA2007

required_modifiers

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Specifies the required modifiers for APIs that should be analyzed	One or more values from the below allowed modifiers table Separate multiple values with a comma (,)	Depends on each rule	CA1802

ALLOWED MODIFIER	SUMMARY
`none`	No modifier requirement
`static` or `Shared`	Must be declared as 'static' ('Shared' in Visual Basic)
`const`	Must be declared as 'const'

ALLOWED MODIFIER	SUMMARY
`readonly`	Must be declared as 'readonly'
`abstract`	Must be declared as 'abstract'
`virtual`	Must be declared as 'virtual'
`override`	Must be declared as 'override'
`sealed`	Must be declared as 'sealed'
`extern`	Must be declared as 'extern'
`async`	Must be declared as 'async'

exclude_extension_method_this_parameter

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Whether to skip analysis for the `this` parameter of extension methods	`true` `false`	`false`	CA1062

null_check_validation_methods

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Names of null check validation methods that validate arguments passed to the method are non-null	Allowed method name formats (separated by `|`): - Method name only (includes all methods with the name, regardless of the containing type or namespace) - Fully qualified names in the symbol's documentation ID format, with an optional `M:` prefix	None	CA1062

additional_string_formatting_methods

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Names of additional string formatting methods	Allowed method name formats (separated by `|`): - Method name only (includes all methods with the name, regardless of the containing type or namespace) - Fully qualified names in the symbol's documentation ID format, with an optional `M:` prefix	None	CA2241

excluded_type_names_with_derived_types

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Names of types, such that the type and all its derived types are excluded for analysis	Allowed symbol name formats (separated by `|`): - Type name only (includes all types with the name, regardless of the containing type or namespace) - Fully qualified names in the symbol's documentation ID format, with an optional `T:` prefix	None	CA1303

excluded_symbol_names

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Names of symbols that are excluded for analysis	Allowed symbol name formats (separated by `|`): - Symbol name only (includes all symbols with the name, regardless of the containing type or namespace) - Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc. - `.ctor` for constructors and `.cctor` for static constructors	None	CA1062 CA1303 CA2000 CA2100 CA2301 CA2302 CA2311 CA2312 CA2321 CA2322 CA2327 CA2328 CA2329 CA2330 CA3001 CA3002 CA3003 CA3004 CA3005 CA3006 CA3007 CA3008 CA3009 CA3010 CA3011 CA3012 CA5361 CA5376 CA5377 CA5378 CA5380 CA5381 CA5382 CA5383 CA5384 CA5387 CA5388 CA5389 CA5390

disallowed_symbol_names

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Names of symbols that are disallowed in the context of the analysis	Allowed symbol name formats (separated by `|`): - Symbol name only (includes all symbols with the name, regardless of the containing type or namespace) - Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc. - `.ctor` for constructors and `.cctor` for static constructors	None	CA1031

Enable a category of rules

1/1/2020 • 2 minutes to read • Edit Online

Analyzer packages may include predefined EditorConfig and rule set files that make it quick and easy to enable a category of rules, such as security or design rules. The Microsoft.CodeAnalysis.FxCopAnalyzers NuGet analyzer package includes both rule sets (starting in version 2.6.2) and EditorConfig files (starting in version 2.9.5). By enabling a specific category of rules, you can identify targeted issues and specific conditions.

> **NOTE**
> Enabling analyzer rules and setting their severity by using an EditorConfig file is supported starting in Visual Studio 2019 version 16.3.

The FxCop analyzer NuGet package includes predefined rule sets and EditorConfig files for the following rule categories:

- All rules
- Dataflow
- Design
- Documentation
- Globalization
- Interoperability
- Maintainability
- Naming
- Performance
- Ported from FxCop
- Reliability
- Security
- Usage

Each of those categories of rules has an EditorConfig or rule set file to:

- enable all the rules in the category (and disable all other rules)
- use each rule's default severity and enablement setting (and disable all other rules)

> **TIP**
> The "all rules" category has an additional EditorConfig or rule set file to disable all rules. Use this file to quickly get rid of any analyzer warnings or errors in a project.

> **TIP**
> If you're migrating from legacy "FxCop" analysis to .NET Compiler Platform-based code analysis, the EditorConfig and rule set files enable you to continue using similar rule configurations to those you used previously.

Predefined EditorConfig files

The predefined EditorConfig files for the Microsoft.CodeAnalysis.FxCopAnalyzers analyzer package are located in the *%USERPROFILE%\.nuget\packages\microsoft.codeanalysis.fxcopanalyzers\<version>\editorconfig* directory. For example, the EditorConfig file to enable all security rules is located at *%USERPROFILE%\.nuget\packages\microsoft.codeanalysis.fxcopanalyzers\<version>\editorconfig\SecurityRulesEnabled\.editorconfig*.

Copy the chosen .editorconfig file to your project's root directory.

Predefined rule sets

The predefined rule set files for the Microsoft.CodeAnalysis.FxCopAnalyzers analyzer package are located in the *%USERPROFILE%\.nuget\packages\microsoft.codeanalysis.fxcopanalyzers\<version>\rulesets* directory. For example, the rule set file to enable all security rules is located at *%USERPROFILE%\.nuget\packages\microsoft.codeanalysis.fxcopanalyzers\<version>\rulesets\SecurityRulesEnabled.ruleset*.

Copy one or more of the rule sets and paste them in the directory that contains your Visual Studio project or directly into **Solution Explorer**.

You can also customize a predefined rule set to your preference. For example, you can change the severity of one or more rules so that violations appear as errors or warnings in the **Error List**.

Set the active rule set

The process for setting the active rule set is a little different depending on whether you have a .NET Core/.NET Standard project or a .NET Framework project.

.NET Core

To make a rule set the active rule set for analysis in .NET Core or .NET Standard projects, manually add the **CodeAnalysisRuleSet** property to your project file. For example, the following code snippet sets `HelloWorld.ruleset` as the active rule set.

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
  ...
  <CodeAnalysisRuleSet>HelloWorld.ruleset</CodeAnalysisRuleSet>
</PropertyGroup>
```

.NET Framework

To make a rule set the active rule set for analysis in .NET Framework projects:

- Right-click on the project in **Solution Explorer** and choose **Properties**.
- In the project property pages, select the **Code Analysis** tab.
- Under **Run this rule set**, select **Browse**, and then select the desired rule set that you copied to the project directory.
- Under **Active rules**, select **Browse**, and then select the desired rule set that you copied to the project directory.

Now you only see rule violations for those rules that are enabled in the selected rule set.

See also

- Analyzers FAQ
- Overview of .NET Compiler Platform analyzers
- Install analyzers

- Configure analyzers
- Use rule sets to group code analysis rules

Overview of code analysis for managed code in Visual Studio

10/4/2019 • 2 minutes to read • Edit Online

Visual Studio can perform code analysis of managed code in two ways: with legacy analysis, also known as FxCop static analysis of managed assemblies, and with the more modern .NET Compiler Platform-based code analyzers. This topic covers legacy analysis. To learn more about .NET Compiler Platform-based code analysis, see Overview of .NET Compiler Platform-based analyzers.

Code analysis for managed code analyzes managed assemblies and reports information about the assemblies, such as violations of the programming and design rules set forth in the .NET Design Guidelines.

The analysis tool represents the checks it performs during an analysis as warning messages. Warning messages identify any relevant programming and design issues and, when it is possible, supply information about how to fix the problem.

> **NOTE**
>
> Legacy analysis (static code analysis) is not supported for .NET Core and .NET Standard projects in Visual Studio. If you run code analysis on a .NET Core or .NET Standard project as part of msbuild, you'll see an error similar to **error : CA0055 : Could not identify platform for <your.dll>**. To analyze code in .NET Core or .NET Standard projects, use code analyzers instead.

IDE (integrated development environment) integration

You can run code analysis on your project manually or automatically.

To run code analysis each time that you build a project, select the option on the project's **Code Analysis** property page. For more information, see How to: Enable and Disable Automatic Code Analysis.

To run code analysis manually on a project, from the menu bar choose **Analyze** > **Run Code Analysis** > **Run Code Analysis on <project>**.

Rule sets

Code analysis rules for managed code are grouped into rule sets. You can use one of the Microsoft standard rule sets, or you can create a custom rule set to fulfill a specific need.

Suppress warnings

Frequently, it is useful to indicate that a warning is non-applicable. This informs the developer, and other people who might review the code later, that a warning was investigated and then either suppressed or ignored.

In-source suppression of warnings is implemented through custom attributes. To suppress a warning, add the attribute `SuppressMessage` to the source code as shown in the following example:

```
[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Design", "CA1039:ListsAreStrongTyped")]
Public class MyClass
{
   // code
}
```

For more information, see Suppress warnings.

> **NOTE**
>
> If you migrate a project to Visual Studio 2017, you might suddenly be faced with a large number of code analysis warnings. If you aren't ready to fix the warnings, you can suppress all of them by choosing **Analyze** > **Run Code Analysis and Suppress Active Issues**.

> **NOTE**
>
> If you migrate a project to Visual Studio 2019, you might suddenly be faced with a large number of code analysis warnings. If you aren't ready to fix the warnings, you can suppress all of them by choosing **Analyze** > **Build and Suppress Active Issues**.

Run code analysis as part of check-in policy

As an organization, you might want to require that all check-ins satisfy certain policies. In particular, you want to make sure that you follow these policies:

- There are no build errors in code being checked in.
- Code analysis is run as part of the most recent build.

You can accomplish this by specifying check-in policies. For more information, see Enhancing Code Quality with Project Check-in Policies.

Team build integration

You can use the integrated features of the build system to run the analysis tool as part of the build process. For more information, see Azure Pipelines.

See also

- Overview of .NET Compiler Platform-based analyzers
- Using Rule Sets to Group Code Analysis Rules
- How to: Enable and Disable Automatic Code Analysis

Walkthrough: Use static code analysis to find code defects

10/17/2019 • 3 minutes to read • Edit Online

In this walkthrough, you'll analyze a managed project for code defects by using legacy code analysis.

This article steps you through the process of using legacy analysis to analyze your .NET managed code assemblies for conformance with the .NET design guidelines.

Create a class library

1. Open Visual Studio and create a new project from the **Class Library (.NET Framework)** template.
2. Name the project **CodeAnalysisManagedDemo**.
3. After the project is created, open the *Class1.cs* file.
4. Replace the existing text in Class1.cs with the following code:

```
using System;

namespace testCode
{
    public class demo : Exception
    {
        public static void Initialize(int size) { }
        protected static readonly int _item;
        public static int item { get { return _item; } }
    }
}
```

5. Save the Class1.cs file.

Analyze the project for code defects

1. Select the CodeAnalysisManagedDemo project in **Solution Explorer**.
2. On the **Project** menu, click **Properties**.

   The CodeAnalysisManagedDemo properties page is displayed.
3. Choose the **Code Analysis** tab.
4. Make sure that **Enable Code Analysis on Build** is selected.
5. From the **Run this rule set** drop-down list, select **Microsoft All Rules**.
4. Make sure that **Run on build** is selected in the **Binary analyzers** section.
5. From the **Active rules** drop-down list, select **Microsoft All Rules**.
6. On the **File** menu, click **Save Selected Items**, and then close the properties pages.
7. On the **Build** menu, click **Build CodeAnalysisManagedDemo**.

   The CodeAnalysisManagedDemo project build warnings are shown in the **Error List** and **Output** windows.

Correct the code analysis issues

1. On the **View** menu, choose **Error List**.

   Depending on the developer profile that you chose, you might have to point to **Other Windows** on the **View** menu, and then choose **Error List**.

2. In **Solution Explorer**, choose **Show All Files**.

3. Expand the Properties node, and then open the *AssemblyInfo.cs* file.

4. Use the following tips to correct the warnings:

   CA1014: Mark assemblies with CLSCompliantAttribute: Add the code `[assembly: CLSCompliant(true)]` to the end of the AssemblyInfo.cs file.

   CA1032: Implement standard exception constructors: Add the constructor `public demo (String s) : base(s) { }` to the class `demo`.

   CA1032: Implement standard exception constructors: Add the constructor `public demo (String s, Exception e) : base(s, e) { }` to the class `demo`.

   CA1032: Implement standard exception constructors: Add the constructor `protected demo (SerializationInfo info, StreamingContext context) : base(info, context) { }` to the class demo. You'll also need to add a `using` statement for System.Runtime.Serialization.

   CA1032: Implement standard exception constructors: Add the constructor `public demo () : base() { }` to the class `demo`.

   CA1709: Identifiers should be cased correctly: Change the casing of the namespace `testCode` to `TestCode`.

   CA1709: Identifiers should be cased correctly: Change the name of the member to `Demo`.

   CA1709: Identifiers should be cased correctly: Change the name of the member to `Item`.

   CA1710: Identifiers should have correct suffix: Change the name of the class and its constructors to `DemoException`.

   CA2237: Mark ISerializable types with SerializableAttribute: Add the `[Serializable ()]` attribute to the class `demo`.

   CA2210: Assemblies should have valid strong names: Sign 'CodeAnalysisManagedDemo' with a strong name key:

   a. On the **Project** menu, choose **CodeAnalysisManagedDemo Properties**.

      The project properties appear.

   b. Choose the **Signing** tab.

   c. Select the **Sign the assembly** check box.

   d. In the **Choose a string name key file** list, select **<New>**.

      The **Create Strong Name Key** dialog box appears.

   e. For **Key file name**, enter **TestKey**.

   f. Enter a password, and then choose **OK**.

   g. On the **File** menu, choose **Save Selected Items**, and then close the property pages.

   After you complete all the changes, the Class1.cs file should look like the following:

```
using System;
using System.Runtime.Serialization;

namespace TestCode
{
    [Serializable()]
    public class DemoException : Exception
    {
        public DemoException () : base() { }
        public DemoException(String s) : base(s) { }
        public DemoException(String s, Exception e) : base(s, e) { }
        protected DemoException(SerializationInfo info, StreamingContext context) : base(info, context)
{ }

        public static void Initialize(int size) { }
        protected static readonly int _item;
        public static int Item { get { return _item; } }
    }
}
```

5. Rebuild the project.

Exclude code analysis warnings

1. For each of the remaining warnings, do the following:

   a. Select the warning in the **Error List**.

   b. From the right-click menu (context menu), choose **Suppress** > **In Suppression File**.

2. Rebuild the project.

   The project builds without any warnings or errors.

See also

Code analysis for managed code

Migrate from legacy analysis (FxCop) to source analysis (FxCop analyzers)

3/7/2020 • 2 minutes to read • Edit Online

Source analysis by .NET Compiler Platform ("Roslyn") analyzers replaces legacy analysis for managed code. For newer project templates such as .NET Core and .NET Standard projects, legacy analysis is not available.

Many of the legacy analysis (FxCop) rules have already been rewritten for FxCop analyzers, a set of Roslyn code analyzers. You install FxCop analyzers as a NuGet package that's referenced by the project or solution. FxCop analyzers run source-code based analysis during compiler execution. Analyzer results are reported along with compiler results.

For more information on the differences between legacy analysis and source analysis, see the following:

- Source code analysis versus legacy analysis
- FAQ about FxCop analyzers

To migrate to source analysis, install the FxCop analyzers. Like legacy analysis rule violations, source code analysis violations appear in the Error List window in Visual Studio. In addition, source code analysis violations also show up in the code editor as *squiggles* under the offending code. The color of the squiggle depends on the severity setting of the rule. To see the status of rules ported to the new FxCop analyzers, see Ported and unported rules.

To learn more about how to configure the FxCop analyzers:

- To configure FxCop analyzers, see Configure FxCop analyzers.
- To find out about configuring analyzers using predefined rules with EditorConfig or a rule set file, see Enable a category of rules.

How to: Run legacy code analysis manually for managed code

8/4/2020 • 2 minutes to read • Edit Online

The code analysis tool provides information to you about possible defects in your source code. You can run code analysis automatically with each build of a code project, and you can also run code analysis manually. The rules that are checked when code analysis is run are specified on the Code Analysis page of the project property pages. For more information, see How to: Configure Code Analysis for a Managed Code Project.

To run code analysis manually

1. If you are on Visual Studio 2019 version 16.5 or later, execute the following command on command prompt before starting Visual Studio:

```
set EnableLegacyCodeAnalysis = true
```

2. In **Solution Explorer**, click the project.
3. On the **Analyze** menu, click **Run Code Analysis on** *Project Name*.

How to: Configure legacy analysis for managed code

8/4/2020 • 2 minutes to read • Edit Online

In Visual Studio, you can choose from a list of code analysis rule sets to apply to a managed code project. By default, the **Microsoft Minimum Recommended Rules** rule set is selected, but you can apply a different rule set if desired. Rule sets can be applied to one or multiple projects in a solution.

> **NOTE**
> This article applies to legacy analysis and not to .NET Compiler Platform-based code analyzers.

Configure a rule set for a .NET Framework project

1. Open the **Code Analysis** tab on the project's property pages. You can do this in either of the following ways:
   - In **Solution Explorer**, select the project. On the menu bar, select **Analyze** > **Configure Code Analysis** > **For <projectname>**.
   - Right-click the project in **Solution Explorer** and select **Properties**, and then select the **Code Analysis** tab.
2. In the **Configuration** and **Platform** lists, select the build configuration and target platform.
3. To run code analysis every time the project is built using the selected configuration, select **Enable Code Analysis on Build**. You can also run code analysis manually by selecting **Analyze** > **Run Code Analysis** > **Run Code Analysis on <projectname>**.
3. To run code analysis every time the project is built using the selected configuration, select **Run on build** in the **Binary analyzers** section. You can also run legacy code analysis manually, see How to: Run Legacy Code Analysis Manually for Managed Code for more details.
4. To view warnings from generated code, clear the **Suppress results from generated code** check box.

   > **NOTE**
   > This option does not suppress code analysis errors and warnings from generated code when the errors and warnings appear in forms and templates. You can both view and maintain the source code for a form or a template, and it won't be overwritten.

5. In the **Run this rule set** list, do one of the following:
5. In the **Active rules** list, do one of the following:

- Select the rule set that you want to use.
- Select **<Browse>** to find an existing custom rule set that is not in the list.
- Define a custom rule set.

Specify rule sets for multiple projects in a solution

By default, all the managed projects of a solution are assigned the *Microsoft Minimum Recommended Rules* code

analysis rule set. You can change the rule sets that are assigned to the projects of a solution in the **Properties** dialog box for the solution.

1. Open the solution in Visual Studio.
2. On the **Analyze** menu, select **Configure Code Analysis for Solution**.
3. If necessary, expand **Common Properties**, and then select **Code Analysis Settings**.
4. You can specify a rule set for one or more projects:
   - To specify a rule set for an individual project, select the project name.
   - To specify a rule set for multiple projects, hold down **Ctrl** and select the project names.
   - To specify all the projects in the solution, hold down **Shift** and click in the project list.
5. Select the **Rule Set** field of a project, and then select the name of the rule set that you want to apply.

See also

- Code analysis rule set reference

How to: Enable and disable binary code analysis for managed code

8/4/2020 • 2 minutes to read • Edit Online

You can configure legacy code analysis (binary analysis) to run after each build of a managed code project. You can also have different settings for each build configuration, for example, debug and release.

> **NOTE**
>
> Legacy analysis is not available for newer project types such as .NET Core and .NET Standard apps. These projects use .NET Compiler Platform-based code analyzers to analyze code, both live and at build time. For information about disabling source code analysis in these projects, see How to disable source code analysis.

To enable or disable legacy code analysis:

1. In **Solution Explorer**, right-click the project, and then choose **Properties**.
2. In the properties dialog box for the project, choose the **Code Analysis** tab.
3. Specify the build type in **Configuration** and the target platform in **Platform**. (Non-.NET Core/.NET Standard projects only.)
4. To enable or disable automatic code analysis, select or clear the **Enable Code Analysis on Build** check box.
4. To enable or disable automatic code analysis, select or clear the **Run on build** check box in the **Binary analyzers** section.

NOTE

Disabling binary code analysis on build does not affect .NET Compiler Platform-based code analyzers, which always execute at build if you installed them as a NuGet package. For information about disabling analysis from these analyzers, see How to disable source code analysis.

How to: Customize the Code Analysis Dictionary

8/4/2020 • 4 minutes to read • Edit Online

Code Analysis uses a built-in dictionary to check identifiers in your code for errors in spelling, grammatical case, and other naming conventions of the .NET design guidelines. You can create a custom dictionary Xml file to add, remove, or modify terms, abbreviations, and acronyms to the built-in dictionary.

For example, suppose your code contained a class named **DoorKnokker**. Code Analysis would identify the name as a compound of two words: **door** and **knokker**. It would then raise a warning that **knokker** was not spelled correctly. To force code analysis to recognize the spelling, you can add the term **knokker** to the custom dictionary.

To create a custom dictionary

Create a file that is named **CustomDictionary.xml**.

Define your custom words by using the following XML structure:

```
<Dictionary>
      <Words>
         <Unrecognized>
            <Word>knokker</Word>
         </Unrecognized>
         <Recognized>
            <Word></Word>
         </Recognized>
         <Deprecated>
            <Term PreferredAlternate=""></Term>
         </Deprecated>
         <Compound>
            <Term CompoundAlternate=""></Term>
         </Compound>
         <DiscreteExceptions>
            <Term></Term>
         </DiscreteExceptions>
      </Words>
      <Acronyms>
         <CasingExceptions>
            <Acronym></Acronym>
         </CasingExceptions>
      </Acronyms>
   </Dictionary>
```

Custom Dictionary Elements

You can modify the behavior of the Code Analysis dictionary by adding terms as the inner text of the following elements in the custom dictionary:

- Dictionary/Words/Recognized/Word
- Dictionary/Words/Unrecognized/Word
- Dictionary/Words/Deprecated/Term[@PreferredAlternate]
- Dictionary/Words/Compound/Term[@CompoundAlternate]
- Dictionary/Words/DiscreteExceptions/Term

- Dictionary/Acronyms/CasingExceptions/Acronym

Dictionary/Words/Recognized/Word

To include a term in the list of terms that code analysis identifies as correctly spelled, add the term as the inner text of a Dictionary/Words/Recognized/Word element. Terms in Dictionary/Words/Recognized/Word elements are not case-sensitive.

Example

```
<Dictionary>
      <Words>
         <Recognized>
            <Word>knokker</Word>
            ...
         </Recognized>
         ...
      </Words>
      ...
</Dictionary>
```

Terms in Dictionary/Words/Recognized nodes are applied to the following code analysis rules:

- CA1701: Resource string compound words should be cased correctly
- CA1702: Compound words should be cased correctly
- CA1703: Resource strings should be spelled correctly
- CA1704: Identifiers should be spelled correctly
- CA1709: Identifiers should be cased correctly
- CA1726: Use preferred terms
- CA2204: Literals should be spelled correctly

Dictionary/Words/Unrecognized/Word

To exclude a term from the list of terms that code analysis identifies as correctly spelled, add the term to exclude as the inner text of a Dictionary/Words/Unrecognized/Word element. Terms in Dictionary/Words/Unrecognized/Word elements are not case-sensitive.

Example

```
<Dictionary>
      <Words>
         <Unrecognized>
            <Word>meth</Word>
            ...
         </Unrecognized>
         ...
      </Words>
      ...
</Dictionary>
```

Terms in the Dictionary/Words/Unrecognized node are applied to the following code analysis rules:

- CA1701: Resource string compound words should be cased correctly
- CA1702: Compound words should be cased correctly
- CA1703: Resource strings should be spelled correctly

- CA1704: Identifiers should be spelled correctly
- CA1709: Identifiers should be cased correctly
- CA1726: Use preferred terms
- CA2204: Literals should be spelled correctly

Dictionary/Words/Deprecated/Term[@PreferredAlternate]

To include a term in the list of terms that code analysis identifies as deprecated, add the term as the inner text of a Dictionary/Words/Deprecated/Term element. A deprecated term is a word that is spelled correctly but should not be used.

To include a suggested alternate term in the warning, specify the alternate in the PreferredAlternate attribute of the Term element. You can leave the attribute value empty if you do not want to suggest an alternate.

- The deprecated term in Dictionary/Words/Deprecated/Term element is not case-sensitive.
- The PreferredAlternate attribute value is case-sensitive. Use Pascal case for compound alternates.

Example

```
<Dictionary>
      <Words>
         <Deprecated>
            <Term PreferredAlternate="LogOn">login</Term>
            ...
         </Deprecated>
         ...
      </Words>
      ...
</Dictionary>
```

Terms in the Dictionary/Words/Deprecated node are applied to the following code analysis rules:

- CA1701: Resource string compound words should be cased correctly
- CA1702: Compound words should be cased correctly
- CA1703: Resource strings should be spelled correctly
- CA1704: Identifiers should be spelled correctly
- CA1726: Use preferred terms

Dictionary/Words/Compound/Term[@CompoundAlternate]

The built-in dictionary identifies some terms as single, discrete terms rather than a compound term. To include a term in the list of terms that code analysis identifies as a compound word and to specify the correct casing of the term, add the term as the inner text of a Dictionary/Words/Compound/Term element. In the CompoundAlternate attribute of the Term element, specify the individual words that make up the compound term by capitalizing the first letter of the individual words (Pascal case). Note that the term specified in the inner text is automatically added to the Dictionary/Words/DiscreteExceptions list.

- The compound term in Dictionary/Words/Compound/Term element is not case-sensitive.
- The CompoundAlternate attribute value is case-sensitive. Use Pascal case for compound alternates.

Example

```
<Dictionary>
      <Words>
         <Compound>
            <Term CompoundAlternate="CheckBox">checkbox</Term>
            ...
         </Compound>
         ...
      </Words>
      ...
</Dictionary>
```

Terms in the Dictionary/Words/Compound node are applied to the following code analysis rules:

- CA1701: Resource string compound words should be cased correctly
- CA1702: Compound words should be cased correctly
- CA1703: Resource strings should be spelled correctly
- CA1704: Identifiers should be spelled correctly

Dictionary/Words/DiscreteExceptions/Term

To exclude a term in the list of terms that code analysis identifies as a single, discrete word when the term is checked by the casing rules for compound words, add the term as the inner text of a Dictionary/Words/DiscreteExceptions/Term element. The term in Dictionary/Words/DiscreteExceptions/Term element is not case-sensitive.

Example

```
<Dictionary>
      <Words>
         <DiscreteExceptions>
            <Term>checkbox</Term>
            ...
         </DiscreteExceptions>
         ...
      </Words>
      ...
</Dictionary>
```

Terms in the Dictionary/Words/DiscreteExceptions node are applied to the following code analysis rules:

- CA1701: Resource string compound words should be cased correctly
- CA1702: Compound words should be cased correctly

Dictionary/Acronyms/CasingExceptions/Acronym

To include an acronym in the list of terms that code analysis identifies as correctly spelled and to indicate how the acronym when the term is checked by the casing rules for compound words, add the term as the inner text of a Dictionary/Acronyms/CasingExceptions/Acronym element. The acronym in the Dictionary/Acronyms/CasingExceptions/Acronym element is case-sensitive.

Example

```
<Dictionary>
      <Acronyms>
         <CasingExceptions>
            <Acronym>NESW</Acronym>   <!-- North East South West -->
            ...
         </CasingExceptions>
         ...
      </Acronyms>
      ...
</Dictionary>
```

Terms in the Dictionary/Acronyms/CasingExceptions node are applied to the following code analysis rules:

- CA1709: Identifiers should be cased correctly

To apply a custom dictionary to a project

1. In **Solution Explorer**, use one of the following procedures:
   - To add a dictionary to a single project, right-click the project name and then click **Add Existing Item**. Specify the file in the **Add Existing Item** dialog box.
   - To add a dictionary that is shared among two or more projects, locate the file to share in the **Add Existing Item** dialog box, click the down arrow on the **Add** button and then click **Add As Link**.
2. In **Solution Explorer**, right-click the **CustomDictionary.xml** file name and click **Properties**.
3. From the **Build Action** list, select **CodeAnalysisDictionary**.
4. From the **Copy to Output Directory** list, select **Do not copy**.

How to: Suppress code analysis warnings for generated code

8/4/2020 • 2 minutes to read • Edit Online

Generated code includes code that's added to your project by managed code compilers or third-party tools. You might want to see the rule violations that code analysis discovers in generated code. However, since you can't view and maintain the code that contains the violations, you might not want to see them.

The **Suppress results from generated code** check box on the code analysis property page of a project enables you to select whether to show code analysis warnings from code generated by a third-party tool.

> **NOTE**
>
> This option does not suppress code analysis errors and warnings from generated code when the errors and warnings appear in forms and templates. You can both view and maintain the source code for a form or a template.

To suppress warnings for generated code in a project

1. Right-click the project in **Solution Explorer** and then click **Properties**.
2. Choose the **Code Analysis** tab.
3. Select the **Suppress results from generated code** check box.

> **IMPORTANT**
>
> You can only suppress warnings from legacy analysis. The property page with the setting has been deprecated and will be removed in a future product release. Currently, you cannot suppress code analysis warnings from analyzers.

Code Analysis Policy Errors

1/1/2020 • 3 minutes to read • Edit Online

The following errors occur if the code analysis policy is not satisfied at check-in:

The Code Analysis settings for one or more projects are not compatible with Code Analysis policy.

The code analysis requirements checking in to the project source control was not met for one or more code projects. This error can be caused by one or more of the following conditions:

- Code Analysis is not enabled on build for all of the projects in the solution.
- The local rule set for the project in Visual Studio has a less restrictive **Action** setting than the project rule set for example, a rule that is set to **Action**=**Error** on the server has its **Action** set to **Warning** or **None** in the rule set being run in Visual Studio).
- The rule set specified in Visual Studio does not contain all of the rules that are specified in the rule set specified in the Code Analysis check-in policy for the project.

The Code Analysis policy failed. There are errors in project {0} or the build is not up to date.

Either the build contains errors or the errors were fixed, but code analysis was not performed after the fix.

Check-in failed. The Code Analysis Policy requires that you check in through Visual Studio with an open solution.

The code analysis policy requires that all files being checked in must be in the currently open solution. To correct this error, open the solution that contains the file to be checked in.

Not all files in the pending check-in are within the currently open solution.

The code analysis policy requires that all files being checked in must be in the currently open solution. This error is raised when there is an open solution, but some files in the "pending check in" view are not part of the currently opened solution. To correct this error, open the solution that contains the file to be checked in.

The version of '{0}' is not correct. The strong-name specified in the policy is '{1}'.

This error applies to .NET projects. A rule .dll required by the code analysis policy exists on the local computer, but the version/public key does not match. To correct this error, the policy creator must update the .dlls in *C:\Program Files\Microsoft Visual Studio 8\Team Tools\Static Analysis Tools\FxCop\Rules* directory on their computer.

'{0}' assembly specified in the policy does not exist.

This error applies to .NET projects. A rule required by the code analysis policy does not have the corresponding dll installed on the client computer. To correct this error, the policy creator must update the dll in *C:\Program Files\Microsoft Visual Studio 8\Team Tools\Static Analysis Tools\FxCop\Rules* directory on their computer.

Project {0} rule settings are not in conformance with Code Analysis policy.

This error applies to .NET projects. The managed code rules settings are not as strict as the policy requires. To correct this error, the client setting must be the same or stricter than the policy requirement on the server.

Code Analysis is not enabled on active configuration. Switch to configuration {0} and build project {1} before checking in.

In Visual Studio, the active configuration does not have code analysis enabled, but there is at least one code analysis enabled.

You must enable Code Analysis for managed binaries in project {0} properties and build before checking in.

This error applies to Visual C++ .NET applications. The policy requires managed code analysis to be performed, but it is not enabled in the current project on the client.

You must enable Code Analysis in project {0} properties and build before checking in.

This error applied to Visual Studio projects and web projects. The policy requires managed code analysis to be performed, but it is not enabled in the current project on the client.

You must enable C/C++ Code Analysis in project {0} properties and build before checking in.

This error applies to unmanaged projects. The code analysis policy requires Code Analysis for C/C++, but it is not enabled in the current project on the client.

See also

- Code Analysis Application Errors

Code analysis application errors

8/4/2020 • 2 minutes to read • Edit Online

This section is a reference of the error messages that are generated by the managed code analysis tool.

In this section

CODE	DESCRIPTION
CA0001	An exception was raised within managed code analysis tool that does not indicate an expected error condition.
CA0051	No rules were selected.
CA0052	No targets were selected to analyze.
CA0053	Rule assembly could not be loaded.
CA0054	A custom rule assembly has invalid XML resources.
CA0055	Could not load file:<path>
CA0056	A project file has an incorrect version of the analysis tool.
CA0057	Violations cannot be mapped to the current set of targets and rules.
CA0058	Unable to load assemblies referenced.
CA0059	Command-line switch error.
CA0060	Unable to load assemblies referenced indirectly.
CA0061	The rule '*RuleId*' could not be found.
CA0062	The rule '*RuleId*' referenced in rule set '*RuleSetName*' could not be found.
CA0063	Failed to load rule set file or one of its dependent rule set files.
CA0064	No analysis was performed because the specified rule set did not contain any FxCop rules.
CA0065	Unsupported metadata construct: Type '*TypeName*' contains both a property and a field with the same name '*PropertyFieldName*'
CA0066	The value '*VersionID*' provided to the **/targetframeworkversion** is not a recognized version.

CODE	DESCRIPTION
CA0067	Directory not found.
CA0068	Debug information could not be found for target assembly *'AssemblyName'*.
CA0069	Using Alternate Platform. *FrameworkVersion1* could not be found. Using *FrameworkVersion2* instead. For best analysis results, ensure that the correct framework version is installed.
CA0070	Cannot load assembly or type due to security permissions.
CA0501	Unable to read output report.
CA0502	Unsupported language.
CA0503	The property is deprecated. Use the superseding property
CA0504	Rule directory was ignored because it does not exist
CA0505	The property is deprecated. Use the superseding property
FxCopCmd Errors	Managed code analysis errors.

Related sections

- Code Analysis Policy Errors
- Analyzing Managed Code Quality

CA0001

1/1/2020 • 2 minutes to read • Edit Online

An exception was raised within managed code analysis tool that does not indicate an expected error condition.

This error is generated in the following cases:

- A defect in a custom rule

  In this case, the error will report the rule and the target. A sample error looks similar to the following:

  Internal Error CA0001: Rule=Microsoft.Usage#CA2214, Target=B..ctor() : Object reference not set to an instance of an object.

  For more diagnostic information, view the CodeAnalysisReport.xml in the \obj folder. The report lists the exception type, stack, type, message, and stack of all inner exceptions. A sample CodeAnalysisReport.xml report for the previous exception looks similar to the following:

  ```
  <Exception Keyword="CA0001" Kind="Rule" TypeName="DoNotCallOverridableMethodsInConstructors"
  Category="Microsoft.Usage" CheckId="CA2214" Target="B..ctor()">
    <Type>System.NullReferenceException</Type>
    <ExceptionMessage>Object reference not set to an instance of an object.</ExceptionMessage>
    <StackTrace>   at
  Microsoft.FxCop.Rules.Usage.DoNotCallOverridableMethodsInConstructors.CheckCallees(Method method,
  Boolean isCallVirt) in d:\rules\DoNotCallOverridableMethodsInConstructors.cs:line 107 at
  Microsoft.FxCop.Rules.Usage.DoNotCallOverridableMethodsInConstructors.CheckCallees(Method method,
  Boolean isCallVirt) in d:\rules\DoNotCallOverridableMethodsInConstructors.cs:line 128 at
  Microsoft.FxCop.Rules.Usage.DoNotCallOverridableMethodsInConstructors.Check(Member member) in
  d:\rules\DoNotCallOverridableMethodsInConstructors.cs:line 58 at
  Microsoft.FxCop.Engines.Introspection.AnalysisVisitor.CheckMember(Member member, NodeBase target) in
  d:\Engines\Introspection\AnalysisVisitor.cs:line 743</StackTrace>
  </Exception>
  ```

  In the previous report, an exception occurred at line 107 of DoNotCallOverridableMethodsInConstructors.cs file. All other stack traces show the execution path that lead to the exception.

- An unknown defect in the managed code analysis tool

  In this case, the error appears without the Rule, TypeName, or Category attributes in CodeAnalysisReport.xml, and the following message appears on the console:

  Internal Error CA0001: Object reference not set to an instance of an object.

  Try rewriting the line where the error is reported, or several lines of code surrounding that line. If that does not work, contact Microsoft Product Support Services.

See also

- Code Analysis Application Errors

CA0051

2/28/2020 • 2 minutes to read • Edit Online

No rules were selected.

This error indicates that the managed code analysis tool is enabled, but no rules have been selected. To correct this error, select appropriate rules.

See also

- Code Analysis Application Errors

CA0052

1/1/2020 • 2 minutes to read • Edit Online

No targets were selected to analyze.

There are two cases that can generate this error:

- Using the managed code analysis tool from within the IDE might cause this error if CA0001 or CA0055 were the root cause of why no targets were selected when the analysis tool was ready to analyze.
- Using FxCopCmd.exe from the command line might cause this error if it was invoked without a /f or /file switch, or with a /p or /project switch pointing to FxCop project file with no targets listed. For help on FxCopCmd.exe, type FxCopCmd /? on command line.

All other error cases should be accompanied by other messages that indicate the root cause of the error.

See also

- Code Analysis Application Errors

CA0053

1/1/2020 • 2 minutes to read • Edit Online

Rule assembly could not be loaded.

This indicates that a rule assembly could not be loaded by reflection. This could occur if the user has custom rules that were not built correctly, or the analysis tool cannot load the assembly for other reasons, such as the file system access was denied.

See also

Code Analysis Application Errors

CA0054

2/28/2020 • 2 minutes to read • Edit Online

Custom rule assembly has invalid XML resources

This occurs if a custom rule assembly has invalid XML resources describing its rules. The correct format is as follows:

```
 <Rules FriendlyName="Customer Rules">
   <Rule TypeName="[The unqualified type name of the rule]" Category="[A category name such as
Customer.Usage]" CheckId="[An identifier for the rule that is at least unique within the same category]">
     <Name>[Localized version of the type name, this is the rule name that appears in the UI]</Name>
     <Description>[A sentence describing the rule in more detail than the name].</Description>
     <Url>[A URL pointing to a documentation or info about the rule]</Url>
     <Resolution>[Format string for rule messages].</Resolution>
     <Email>[Email Address]</Email>
     <MessageLevel Certainty="[A number from 0 to 100]">[Error or Warning]</MessageLevel>
     <FixCategories>[NonBreaking or Breaking]</FixCategories>
     <Owner>[Rule owner name or group]</Owner>
   </Rule>
 </Rules>
```

See also

Code Analysis Application Errors

CA0055

1/1/2020 • 2 minutes to read • Edit Online

Could not load file:<path>

The code analysis tool was unable to load the target file that was specified for analysis. This occurs if the file is not found or access to the file was denied. Typically, this occurs if the user chose to override the input assembly property in Visual Studio, or invoked fxcopcmd.exe manually.

See also

Code Analysis Application Errors

CA0056

2/28/2020 • 2 minutes to read • Edit Online

A project file has an incorrect version of the analysis tool.

This warning occurs if FxCopCmd.exe is invoked manually with a project file that has an incorrect code analysis tool version.

See also

Code Analysis Application Errors

CA0057

2/28/2020 • 2 minutes to read • Edit Online

Violations cannot be mapped to the current set of targets and rules

This error occurs if FxCopCmd.exe is invoked manually with a project file or an /imported XML report that has violations that cannot be mapped to the current set of targets and rules.

Occurs only if analysis is done using the command-line version of the tool.

See also

Code Analysis Application Errors

CA0058

2/28/2020 • 2 minutes to read • Edit Online

Unable to load assemblies referenced.

This error occurs if the analysis tool is unable to load assemblies referenced by the assembly under analysis. As result, this error might cause CA0001 in other places because many unexpected states can arise if this occurs. If you are using the managed code analysis tool from within Visual Studio, the following are some of the reasons that might cause CA0058:

- The input assembly was overridden and now points to an assembly that references other assemblies outside the list of references in the Visual Studio project file.
- If a project is modified to invoke a custom build step before FxCopCmd.exe runs and the custom build process adds more references to the input assembly, error CA0058 is generated.
- In C++, it is possible to reference assemblies using ForcedUsing in VCPROJ, overriding the command-line to pass / ForcedUsing or adding #using <Some.dll> in source only.

In all cases, the resolution for this issue is to add the missing reference to the Visual Studio project itself.

See also

- Code Analysis Application Errors

CA0059

1/1/2020 • 2 minutes to read • Edit Online

Command line switch error.

This error occurs if a required command line switch is missing, for example,

/out or /console switch is not specified. This error also occurs if there is an invalid switch, such as a bad time out value.

See also

Code Analysis Application Errors

CA0060

2/28/2020 • 2 minutes to read • Edit Online

Unable to load assemblies referenced indirectly.

This warning occurs if the analysis tool is unable to load assemblies that are indirectly referenced by the assembly under analysis. An "indirect reference" refers to a reference assembly that one of your analysis assembly's assembly references refers to. For example if code analysis (FxCop) is analyzing assembly A, and assembly "A" references assembly "B", and assembly "B" references assembly "C" but "A" does not reference assembly "C", then assembly "C" is an indirect reference and assembly "B" is a direct reference.

The inability to load assemblies might cause error CA0001 in other places, because unexpected states can result. If you are using the managed code analysis tool from within Visual Studio, the following are some of the reasons that might cause CA0060 warning:

- The input assembly was overridden and now points to an assembly that references other assemblies outside the list of references in the Visual Studio project file.
- If a project is modified to invoke a custom build step before FxCopCmd.exe runs and the custom build process adds more references to the input assembly and warning CA0060 is generated.
- In C++, you can reference assemblies by using ForcedUsing in VCPROJ, overriding the command line to pass /FU or adding #using <Some.dll> in source only.
- A third-party assembly that has some private references to other assemblies that you do not need in order to compile and run your code.

In all cases, the resolution for this issue is to add the missing reference to the Visual Studio project itself.

See also

- Code Analysis Application Errors
- CA0001

CA0061

1/16/2020 • 2 minutes to read • Edit Online

The rule '*RuleId*' could not be found.

This error indicates that the specified rule could not be found.

This warning can be caused by an incorrectly formatted **FxCopCmd.exe /RuleId** option, an incorrectly formatted CodeAnalysisRules property value, or because the specified rule is in a rule assembly that FxCop is not using.

FxCopCmd /RuleId option

Use one of the following formats to specify a rule in the **FxCopCmd.exe /RuleId** option on the FxCopCmd command line:

- **FxCopCmd.exe /RuleId:-** *Category* **#** *RuleId*

  where *Category* is the rule category and *RuleId* is the CheckId of the rule. For example:

  ```
  FxCopCmd /RuleId:-Microsoft.Design#CA2210
  ```

- **FxCopCmd.exe /RuleId:-** *Namespace* **#** *RuleId*

  where *Namespace* is the rule category and *RuleId* is the check id of the rule. For example:

  ```
  FxCopCmd /RuleId:-Microsoft.Rules.Design#CA2210
  ```

MSBuild CodeAnalysisRules property

In Visual Studio code analysis, rules can be specified by using the CodeAnalysisRules property of MSBuild with the following format:

<CodeAnalysisRules>-{*Category*|*Namespace*}#*RuleId*[;...]**</CodeAnalysisRules>**

For example

```
<CodeAnalysisRules>-Microsoft.Design#CA2210;-Microsoft.Rules.Managed.CA1062</CodeAnalysisRules>
```

See also

Code Analysis Application Errors

CA0062

2/28/2020 • 2 minutes to read • Edit Online

The rule '*RuleId*' referenced in rule set '*RuleSetName*' could not be found.

This error indicates that the specified rule couldn't be found.

This error usually occurs because a rule set file was edited by hand. To avoid this error, you can use the Visual Studio rule set editor to configure code analysis rules.

To resolve this issue, make sure that all the check IDs that are specified in your *.ruleset* file are valid. if you are using a non-standard rule set file, make sure that the appropriate rule hint paths are specified in the file.

See also

- Code Analysis Application Errors

CA0063

1/1/2020 • 2 minutes to read • Edit Online

Failed to load rule set file or one of its dependent rule set files.

The specified rule set could not be found, or one of the rule set files included in your rule set could not be found. Make sure that the all the rule sets included in your rule set exist on disk and that the appropriate rule set directories are specified in your project through the **CodeAnalysisRuleSetDirectories** property of MSBuild.

To debug the error, examine your rule set file in a text editor:

1. To find the path to the rule set file, right-click the project in **Solution Explorer**, click **Properties**, and then click **Code Analysis**. Make sure the rule set file is selected as the active rule set. The path to the rule set is listed in the description field.

2. Open the rule set file in a text editor.

3. Examine the **Path** attribute value in all the **Include** elements, and verify that they are valid. Include paths can use environment variables, absolute paths, and relative paths (relative to the current rule set file). For example:

```
<Include Path="%PUBLIC%\Documents\RuleSets\alldesign.ruleset" Action="Default" />
<Include Path="..\alldesign.ruleset" Action="Default" />
<Include Path="C:\Rulesets\alldesign.ruleset" Action="Default" />
```

In some cases, the location of a rule set can be dependent on an MSBuild property. MSBuild properties cannot be referenced from a rule set. To work around this issue, specify additional search paths in the **CodeAnalysisRuleSetDirectories** property of MSBuild. In this scenario, specify only the name of the rule set in the **Path** attribute of the **Include** element.

See also

- Code Analysis Application Errors

CA0064

1/1/2020 • 2 minutes to read • Edit Online

No analysis was performed because the specified rule set did not contain any FxCop rules.

This warning can occur in one of the following situations:

- If you encounter this warning in conjunction with CA0063 warnings, there was a problem loading your rule set file. For more information, see the CA0063 article.
- If you encounter this error in conjunction with CA0062 warnings, it most likely that code analysis was unable to find the assembly or assemblies that contain the rules specified by the rule set. For more information, see the CA0062 article.
- Otherwise, this warning usually occurs when your rule set is empty or all of the rules enabled in a child rule set are disabled. Use the Visual Studio rule set editor to enable some rules in your rule set.

See also

- Code analysis application errors

CA0065

2/28/2020 • 2 minutes to read • Edit Online

Unsupported metadata construct: Type '*TypeName*' contains both a property and a field with the same name '*PropertyFieldName*'

See also

Code Analysis Application Errors

CA0066

2/28/2020 • 2 minutes to read • Edit Online

The value '*VersionID*' provided to the /targetframeworkversion is not a recognized version.

See also

Code Analysis Application Errors

CA0067

2/28/2020 • 2 minutes to read • Edit Online

Directory not found.

The value of the **/directory** option in the FxCopCmd command line was not found.

This warning can occur if you are using the **CodeAnalysisDependentAssemblyPaths** property of MSBuild to specify additional reference assembly search paths and one of those paths does not exist.

- If warning CA0067 appears with CA0058 errors or CA0060 warnings, resolving the other errors usually resolves CA0067.
- If warning CA0067 appears without other errors or warnings, you can usually ignore the warning because the directory was not needed.

See also

Code Analysis Application Errors

CA0068

1/1/2020 • 2 minutes to read • Edit Online

Debug information (symbols) could not be found for target assembly *'AssemblyName'*. For best analysis results, include the .pdb file with debug information for *'AssemblyName'* in the same directory as the target assembly.

The debug information found in .pdb files improves the accuracy of some Code Analysis checks. A missing .pdb file can lead to increased false positives. To enable .pdb generation, open the project properties page. On the **Build** tab, find the **Debug Info** list. The location varies by project type. Make sure that **Debug info** is either set to **full** for debug builds or to **pdb-only** for release builds. It should not be set to **none**.

See also

- Code Analysis Application Errors
- Specify Symbol (.pdb) and Source Files

CA0069

2/28/2020 • 2 minutes to read • Edit Online

UsingAlternatePlatform. *FrameworkVersion1* could not be found. Using *FrameworkVersion2* instead. For best analysis results please ensure that the correct framework version is installed.

See also

- Code Analysis Application Errors

CA0070

2/28/2020 • 2 minutes to read • Edit Online

This error occurs when code analysis analyzes an assembly and encounters permission attributes and at least one of the following conditions is true:

- Code analysis cannot find the assembly that contains the attributes.
- Code analysis does not have permission to load the assembly.
- The assembly that is loaded by code analysis does not contain the attribute.

In most cases, the issue can be resolved by making sure that you have the most recent version of the code analysis tools.

See also

- Code Analysis Application Errors

CA0501

1/1/2020 • 2 minutes to read • Edit Online

Unable to read output report

This occurs when output report cannot be written because of one of the following reasons:

- Disk full
- No permissions to write to directory
- Directory does not exist

See also

Code Analysis Application Errors

CA0502

2/28/2020 • 2 minutes to read • Edit Online

Unsupported language

This error occurs when the user tries to analyze an ASP.NET project that contains one or more pages or classes written in a language other than C# or Visual Basic. To fix this error, remove the file from the project or disable code analysis.

See also

Code Analysis Application Errors

CA0503

1/16/2020 • 2 minutes to read • Edit Online

The property is deprecated. Use the superseding property

Warning CA0503 is generated when a deprecated code analysis property contains a value in the build or project configuration file. To resolve this warning, use the specified property that supersedes it.

See also

Code Analysis Application Errors

CA0504

2/28/2020 • 2 minutes to read • Edit Online

Rule directory was ignored because it does not exist

Warning CA504 is generated when a directory specified in the **CodeAnalysisRuleDirectories** property in a project or build configuration file could not be found. To resolve this issue, enter a valid directory path.

See also

Code Analysis Application Errors

CA0505

1/16/2020 • 2 minutes to read • Edit Online

The deprecated property will be ignored because the superceding property is defined.

Warning CA505 is generated when both the deprecated and superseding code analysis properties have values in a project or build configuration file. To resolve this issue, remove the deprecated property.

See also

- Code Analysis Application Errors
- Using Rule Sets to Group Code Analysis Rules

FxCopCmd tool errors

2/28/2020 • 2 minutes to read • Edit Online

FxCopCmd does not consider all errors to be fatal. If FxCopCmd has sufficient information to perform a partial analysis, it performs the analysis and reports errors that occurred. The error code, which is a 32-bit integer, contains a bitwise combination of numeric values that correspond to errors.

The following table describes the error codes returned by FxCopCmd:

ERROR	NUMERIC VALUE
No errors	0x0
Analysis error	0x1
Rule exceptions	0x2
Project load error	0x4
Assembly load error	0x8
Rule library load error	0x10
Import report load error	0x20
Output error	0x40
Command line switch error	0x80
Initialization error	0x100
Assembly references error	0x200
BuildBreakingMessage	0x400
Unknown error	0x1000000

Analysis error is returned for fatal errors. It indicates that the analysis could not be completed. When applicable, the error code also contains the underlying cause of the fatal error. The following conditions generate fatal errors:

- The analysis could not be performed because of insufficient input.
- The analysis threw an exception that is not handled by FxCopCmd.
- The specified project file could not be found or is corrupted.
- The output option was not specified or the file could not be written.

> **NOTE**
>
> The FxCopCmd return code **Assembly references error** 0x200 by itself is a warning rather than an error. This return code indicates that there are missing indirect references, but that FxCopCmd was able to handle them. The warning means there's a possibility that some analysis results might have been compromised. Treat **Assembly references error** as an error when it is combined with any other return code.

See also

- Code Analysis Application Errors

How to: Create or Update Standard Code Analysis Check-in Policies

8/4/2020 • 2 minutes to read • Edit Online

You can require that code analysis be run on all code projects in an Azure DevOps project by using the code analysis check-in policy. Requiring code analysis can improve the quality of the code that is checked into the code base.

> **NOTE**
>
> This feature is available only if you are using Team Foundation Server.

Code analysis check-in policies are set in the project settings and apply to each code project. Code analysis runs are configured for code projects in the project (.xxproj) file for the code project. Code analysis runs are performed on the local computer. When you enable a code analysis check-in policy, files in a code project that are to be checked in must be compiled after their last edit and a code analysis run that contains, at a minimum, the rules in the project settings must be performed on the computer where the changes have been made.

- For managed code, you set the check-in policy by specifying a *rule set* that contains a subset of the code analysis rules.
- For C/C++ code, in Visual Studio 2017 version 15.6 and earlier, the check-in policy requires that all code analysis rules are run. You can add pre-processor directives to disable specific rules for the individual code projects in your Azure DevOps project. In 15.7 and later, you can use **/analyze:ruleset** to specify which rules to run. For more information, see Using Rule Sets to Specify the C++ Rules to Run.

After you specify a check-in policy for managed code, team members can synchronize their code analysis settings for code projects to the Azure DevOps project policy settings.

To open the check-in policy editor

1. In Team Explorer, right-click the project name, point to **Project Settings**, and then click **Source Control**.
2. In the **Source Control** dialog box, select the **Check-in Policy** tab.
3. Do one of the following:
   - Click **Add** to create a new check-in policy.
   - Double-click the existing **Code Analysis** item in the **Policy Type** list to change the policy.

To set policy options

Select or clear the following options:

OPTION	DESCRIPTION
Enforce check-in to only contain files that are part of current solution.	Code analysis can run only on files specified in solution and project configuration files. This policy guarantees that all code that is part of a solution is analyzed.

OPTION	DESCRIPTION
Enforce C/C++ Code Analysis (/analyze)	Requires that all C or C++ projects be built with the /analyze compiler option to run code analysis before they can be checked in.
Enforce Code Analysis for Managed Code	Requires that all managed projects run code analysis and build before they can be checked in.

To specify a managed rule set

From the **Run this rule set** list, use one of the following methods:

- Select a Microsoft standard rule set.
- Select a custom rule set by clicking **<Select Rule Set from Source Control...>**. Then, type the version control path of the rule set in the source control browser. The syntax of a version control path is:

  $/ `TeamProjectName` **/** `VersionControlPath`

For more information about how to create and implement a custom check-in policy rule set, see Implement Custom Check-in Policies for Managed Code.

See also

- Implement Custom Code Analysis Check-in Policies for Managed Code

Implement Custom Code Analysis Check-in Policies for Managed Code

8/4/2020 • 5 minutes to read • Edit Online

A code analysis check-in policy specifies a set of rules that members of an Azure DevOps project must run on source code before it is checked in to version control. Microsoft provides a set of standard *rule sets* that group code analysis rules into functional areas. *Custom check-in policy rule sets* specify a set of code analysis rules that are specific to a project. A rule set is stored in a .ruleset file.

Check-in policies are set at the Azure DevOps project level and specified by the location of a .ruleset file in the version control tree. There are no restrictions on the version control location of the team policy custom rule set.

Code analysis is configured for the individual code projects in the properties window for each project. A custom rule set for a code project is specified by the physical location of the .ruleset file on the local computer. When a .ruleset file is specified that is located on the same drive as the code project, Visual Studio uses a relative path to the file in the project configuration.

A suggested practice for creating an Azure DevOps project custom rule set is to store the check-in policy .ruleset file in a special folder that is not a part of any code project. If you store the file in a dedicated folder, you can apply permissions that restrict who can edit the rule file, and you can easily move the directory structure that contains the project to another directory or computer.

Create the Project Custom Check-in Rule Set

To create a custom rule set for an Azure DevOps project, you first create a special folder for the check-in policy rule set in **Source Control Explorer**. Then you create the rule set file and add the file to version control. Finally, you specify the rule set as the code analysis check-in policy for the project.

> **NOTE**
>
> To create a folder in an Azure DevOps project, you first must map the project root to a location on the local computer.

To create the version control folder for the check-in policy rule set

1. In Team Explorer, expand the project node, and then click **Source Control**.
2. In the **Folders** pane, right-click the project and then click **New Folder**.
3. In the main Source Control pane, right-click **New Folder**, click **Rename**, and type a name for the rule set folder.

To create the check-in policy rule set

1. On the **File** menu, point to **New**, and then click **File**.
2. In the **Categories** list, click **General**.
3. In the **Templates** list, double-click **Code Analysis Rule Set**.
4. Specify the rules to include in the rule set, and then save the rule set file to the rule set folder that you created.

To add the rule set file to version control

1. In **Source Control Explorer**, right-click the new folder, and then click **Add Items to Folder**.

For more information, see Git and Azure Repos.

2. Click the rule set file that you created, and then click **Finish**.

   The file is added to source control and checked out to you.

3. In the **Source Control Explorer** details window, right-click the file name and then click **Check in Pending Changes**.

4. In the **Check-in** dialog box, you have the option to add a comment and then click **Check In**.

> **NOTE**
> If you have already configured a code analysis check-in policy for your Azure DevOps project and you have selected the **Enforce check-in to only contain files that are part of current solution**, you will trigger a policy failure warning. In the Policy Failure dialog box, select **Override policy failure and continue checkin**. Add a required comment, and then click **OK**.

To specify the rule set file as the check-in policy

1. On the **Team** menu, point to **Project Settings**, and then click **Source Control**.

2. Click **Check-in Policy**, and then click **Add**.

3. In the **Check-in Policy** list, double-click **Code Analysis**, and make sure that the **Enforce Code Analysis for Managed Code** check box is selected.

4. In the **Run this rule set** list, click **<Select Rule Set from Source Control>**.

5. Type the path of the check-in policy rule set file in version control.

   The path must conform to the following syntax:

   $/ `TeamProjectName` **/** `VersionControlPath`

> **NOTE**
> You can copy the path by using one of the following procedures in **Source Control Explorer**:

- In the **Folders** pane, click the folder that contains the rule set file. Copy the version control path of the folder that appears in the **Source** box, and type the name of the rule set file manually.

- In the details window, right-click the rule set file, and then click **Properties**. On the **General** tab, copy the value in **Server Name**.

Synchronize Code Projects to the Check-in Policy Rule Set

You specify a project check-in policy rule set as the code analysis rule set of a code project configuration in the properties dialog box of the code project. If the rule set is located on the same drive as the code project, a relative path is used to specify rule set when the path is selected from the file dialog box. The relative path enables the project properties settings to be portable to other computers that use similar local version control structures.

To specify a project rule set as the rule set of a code project

1. If necessary, retrieve the check-in policy rule set folder and file from version control.

   You can perform this step in **Source Control Explorer** by right-clicking the rule set folder and then clicking **Get Latest Version**.

2. In **Solution Explorer**, right-click the code project, and then click **Properties**.
3. **Click Code Analysis**.
4. If necessary, click the appropriate options in the **Configuration** and **Platform** lists.
5. To run code analysis every time that the code project is built using the specified configuration, select **Enable Code Analysis on Build**.
5. To run code analysis every time that the code project is built using the specified configuration, select **Run on build** in the **Binary analyzers** section.
6. In the **Run this rule set** list, click **<Browse>**.
7. Select the local version of the check-in policy rule set file.

How to: Enforce maintainable code with a code analysis check-in policy

8/4/2020 • 2 minutes to read • Edit Online

Developers can use the Code Metrics tool to measure the complexity and maintainability of their code, but you cannot invoke Code Metrics as part of a check-in policy. However, you can enable Code Analysis rules that verify the compliance of your code with code metrics standards, and enforce the rules through check-in policies. For more information about code metrics, see Code metrics values.

You can enable the Depth of Inheritance, Class Coupling, Maintainability Index, and Complexity rules to enforce maintainable code through a Code Analysis check-in policy. All four of these rules are found under the "Maintainability Rules" category in the Code Analysis policy editor.

Administrators of version control for Team Foundation can add the Code Analysis Maintainability Rules to the check-in policy requirements. These check-in policies require developers to run Code Analysis based on these rule changes before initiating a check-in.

To open the Code Analysis Policy editor

1. In **Team Explorer**, right-click the project, click **Project Settings**, and then click **Source Control**.

   The **Source Control** dialog box appears.

2. On the **Check-in Policy** tab, and click **Add**.

   The **Add Check-in Policy** dialog box appears.

3. In the **Check-in Policy** list, select the **Code Analysis** check box, and then click **OK**.

   The **Code Analysis Policy Editor** dialog box appears.

To enable code analysis maintainability rules

1. In the **Code Analysis Policy Editor** dialog box, under **Rule Settings**, expand the **Maintainability Rules** node.

2. Select the check boxes for the following rules:

   - Depth of Inheritance: **CA1501 AvoidExcessiveInheritance** - Threshold: Warning at more than 5 levels deep
   - Complexity: **CA1502 AvoidExcessiveComplexity** - Threshold: Warning at more than 25
   - Maintainability Index: **CA1505 AvoidUnmaintainableCode** - Threshold: Warning at fewer than 20
   - Class Coupling: **CA1506 AvoidExcessiveClassCoupling** - Threshold: Warning at more than 80 for a class and more than 30 for a method

     In addition, if you want a rule violation to prevent a successful build, select the **Treat Warning As An Error** check box next to the rule description.

3. Click **OK**. The new check-in policy now applies to future check-ins.

See also

- Code metrics values
- Creating and using code analysis check-in policies

How to: Synchronize Code Project Rule Sets with an Azure DevOps Project Check-in Policy

8/4/2020 • 2 minutes to read • Edit Online

You synchronize the code analysis settings for code projects to the check-in policy for the Azure DevOps project by specifying a rule set that contains at least the rules that are specified in the rule set for the check-in policy. Your developer lead can inform you of the name and location of the rule set for the check-in policy. You can use one of the following options to ensure that code analysis for the project uses the correct set of rules:

- If the check-in policy uses one of the Microsoft built-in rule sets, open the properties dialog box for the code project, display the Code Analysis page, and select the rule set. The Microsoft standard rule sets are automatically installed with Visual Studio are set to read-only and should not be edited. If the rule sets are not edited, the rules in the policy and local rule sets are guaranteed to match.
- If the check-in policy uses a custom rule set, perform a get operation on the rule set file in version control to create a local copy. Then specify that local location in the code analysis settings for the code project. The rules are guaranteed to match if the rule set for the check-in policy is up to date.

  If you map the version control location to a local folder that is in the same relationship to the Azure DevOps project root as your code project, the location of the rule is set by using a relative path. The relative path ensures that the code project setting for code analysis can be moved to other computers.
- Customize a copy of the rule set for the check-in policy for a code project. Make sure that the new rule set contains all the rules in the check-in policy and any other rules that you want to include. You must make sure that your rule set includes all the rules in the rule set for the check-in policy.

To specify a Microsoft standard rule set

1. In **Solution Explorer**, right-click the code project, and then click **Properties**.
2. Click **Code Analysis**.
3. In the **Run this rule set** list, select the check-in policy rule set.
3. In the **Active rules** list, select the check-in policy rule set.

To specify a custom check-in policy rule set

1. If necessary, perform a get operation on the rule set file that specifies the check-in policy.
2. In **Solution Explorer**, right-click the code project, and then click **Properties**.
3. Click **Code Analysis**.
4. In the **Run this rule set** list, click **<Browse>**.
4. In the **Active rules** list, click **<Browse>**.
5. In the **Open** dialog box, specify the check-in policy rule set file.

To create a custom rule set for a code project

For information about creating a custom rule set, see Customize a rule set.

Version Compatibility for Code Analysis Check-In Policies

1/1/2020 • 2 minutes to read • Edit Online

If you must evaluate and author code analysis check-in policies using different versions of Team Explorer, you must know the differences in how Visual Studio Team System 2008 Team Foundation Server and Team Foundation Server 2010 evaluate check-in policies.

Version Compatibility for Evaluating Check-In Policies

- When code analysis check-in policies are evaluated in Team System 2008 Team Foundation Server, any rules that existed in Team Foundation Server 2010 but do not exist in Team System 2008 Team Foundation Server are ignored.
- When code analysis check-in policies are evaluated in Team Foundation Server 2010, all new rules that are exclusive to Team System 2008 Team Foundation Server are ignored.
- If the code analysis check-in policy specifies rules assemblies, Team System 2008 Team Foundation Server ignores all rules that are specified by assemblies that it does not recognize.
- If the code analysis check-in policy specifies rules assemblies that Team Foundation Server 2010 does not recognize, a message is displayed.

Version Compatibility for Authoring Check-In Policies

- If you created a code analysis check-in policy by using the Team System 2008 Team Foundation Server version of Team Explorer, you cannot use the Team Foundation Server 2010 version of Team Explorer to modify it. And also, Team Foundation Server 2010 cannot evaluate the policy.
- If you created a code analysis check-in policy by using Team Explorer in Team Foundation Server 2010, you can use Team Explorer in Team System 2008 Team Foundation Server to modify it, and the policy can also be evaluated by Team System 2008 Team Foundation Server. After you modify the policy by using Team Explorer in Team System 2008 Team Foundation Server, you can no longer edit the policy by using Team Explorer in Team Foundation Server 2010. Team Foundation Server 2010 can evaluate the policies without problems with mismatched strong names.
- To create a code analysis check-in policy with rule settings that apply for both Team Foundation Server 2010 and Team System 2008 Team Foundation Server, you must create the policy in Team Foundation Server 2010, make all the changes needed, and save the policy. If the changes to rules exist only in Team System 2008 Team Foundation Server, you modify and save the policy in Team System 2008 Team Foundation Server.

  After you save the policy in Team System 2008 Team Foundation Server, you can no longer change settings for rules that exist in Team Foundation Server 2010 only.

Use rule sets to group code analysis rules

2/19/2020 • 2 minutes to read • Edit Online

When you configure code analysis in Visual Studio, you can choose from a list of built-in *rule sets*. A rule set is a grouping of code analysis rules that identify targeted issues and specific conditions for that project. For example, you can apply a rule set that's designed to scan code for publicly available APIs. You can also apply a rule set that includes all the available rules.

You can customize a rule set by adding or deleting rules or by changing rule severities to appear as either warnings or errors in the **Error List**. Customized rule sets can fulfill a need for your particular development environment. When you customize a rule set, the rule set editor provides search and filtering tools to help you in the process.

Rule sets are available for managed code analysis, legacy analysis of managed code, and C++ code analysis.

Rule set format

A rule set is specified in XML format in a *.ruleset* file. Rules, which consist of an ID and an *action*, are grouped by analyzer ID and namespace in the file.

The contents of a *.ruleset* file looks similar to this XML:

```
<RuleSet Name="Rules for Hello World project" Description="These rules focus on critical issues for the Hello World app." ToolsVersion="10.0">
  <Localization ResourceAssembly="Microsoft.VisualStudio.CodeAnalysis.RuleSets.Strings.dll" ResourceBaseName="Microsoft.VisualStudio.CodeAnalysis.RuleSets.Strings.Localized">
    <Name Resource="HelloWorldRules_Name" />
    <Description Resource="HelloWorldRules_Description" />
  </Localization>
  <Rules AnalyzerId="Microsoft.Analyzers.ManagedCodeAnalysis" RuleNamespace="Microsoft.Rules.Managed">
    <Rule Id="CA1001" Action="Warning" />
    <Rule Id="CA1009" Action="Warning" />
    <Rule Id="CA1016" Action="Warning" />
    <Rule Id="CA1033" Action="Warning" />
  </Rules>
  <Rules AnalyzerId="Microsoft.CodeQuality.Analyzers" RuleNamespace="Microsoft.CodeQuality.Analyzers">
    <Rule Id="CA1802" Action="Error" />
    <Rule Id="CA1814" Action="Info" />
    <Rule Id="CA1823" Action="None" />
    <Rule Id="CA2217" Action="Warning" />
  </Rules>
</RuleSet>
```

TIP

It's easier to edit a rule set in the graphical **Rule Set Editor** than by hand.

Specify a rule set for a project

The rule set for a project is specified by the **CodeAnalysisRuleSet** property in the Visual Studio project file. For example:

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
  ...
  <CodeAnalysisRuleSet>HelloWorld.ruleset</CodeAnalysisRuleSet>
</PropertyGroup>
```

See also

- Code analysis rule set reference

How to: Configure legacy analysis for managed code

8/4/2020 • 2 minutes to read • Edit Online

In Visual Studio, you can choose from a list of code analysis rule sets to apply to a managed code project. By default, the **Microsoft Minimum Recommended Rules** rule set is selected, but you can apply a different rule set if desired. Rule sets can be applied to one or multiple projects in a solution.

> **NOTE**
> This article applies to legacy analysis and not to .NET Compiler Platform-based code analyzers.

Configure a rule set for a .NET Framework project

1. Open the **Code Analysis** tab on the project's property pages. You can do this in either of the following ways:
   - In **Solution Explorer**, select the project. On the menu bar, select **Analyze** > **Configure Code Analysis** > **For <projectname>**.
   - Right-click the project in **Solution Explorer** and select **Properties**, and then select the **Code Analysis** tab.
2. In the **Configuration** and **Platform** lists, select the build configuration and target platform.
3. To run code analysis every time the project is built using the selected configuration, select **Enable Code Analysis on Build**. You can also run code analysis manually by selecting **Analyze** > **Run Code Analysis** > **Run Code Analysis on <projectname>**.
3. To run code analysis every time the project is built using the selected configuration, select **Run on build** in the **Binary analyzers** section. You can also run legacy code analysis manually, see How to: Run Legacy Code Analysis Manually for Managed Code for more details.
4. To view warnings from generated code, clear the **Suppress results from generated code** check box.

   > **NOTE**
   > This option does not suppress code analysis errors and warnings from generated code when the errors and warnings appear in forms and templates. You can both view and maintain the source code for a form or a template, and it won't be overwritten.

5. In the **Run this rule set** list, do one of the following:
5. In the **Active rules** list, do one of the following:

- Select the rule set that you want to use.
- Select **<Browse>** to find an existing custom rule set that is not in the list.
- Define a custom rule set.

Specify rule sets for multiple projects in a solution

By default, all the managed projects of a solution are assigned the *Microsoft Minimum Recommended Rules* code analysis rule set. You can change the rule sets that are assigned to the projects of a solution in the **Properties** dialog box for the solution.

1. Open the solution in Visual Studio.
2. On the **Analyze** menu, select **Configure Code Analysis for Solution**.
3. If necessary, expand **Common Properties**, and then select **Code Analysis Settings**.
4. You can specify a rule set for one or more projects:
   - To specify a rule set for an individual project, select the project name.
   - To specify a rule set for multiple projects, hold down **Ctrl** and select the project names.
   - To specify all the projects in the solution, hold down **Shift** and click in the project list.
5. Select the **Rule Set** field of a project, and then select the name of the rule set that you want to apply.

See also

- Code analysis rule set reference

Customize a rule set

8/4/2020 • 3 minutes to read • Edit Online

You can create a custom rule set to meet specific project needs for code analysis.

Create a custom rule set from an existing rule set

To create a custom rule set, you can open a built-in rule set in the **rule set editor**. From there, you can add or remove specific rules, and you can change the action that occurs when a rule is violated—for example, show a warning or an error.

1. In **Solution Explorer**, right-click the project and then select **Properties**.
2. On the **Properties** pages, select the **Code Analysis** tab.
3. In the **Run this rule set** drop-down list, do one of the following:
3. In the **Active rules** drop-down list, do one of the following:

- Select the rule set that you want to customize.

  - or -

- Select **<Browse>** to specify an existing rule set that is not in the list.

4. Select **Open** to display the rules in the rule set editor.

> **NOTE**
>
> If you have a .NET Core or .NET Standard project, the process is a little different because there's no **Code Analysis** property tab. Follow the steps to copy a predefined rule set to your project and set it as the active rule set. After you've copied over a rule set, you can edit it in the Visual Studio rule set editor by opening it from **Solution Explorer**.

Create a new rule set

You can create a new rule set file from the **New File** dialog:

1. Select **File** > **New** > **File**, or press **Ctrl**+**N**.
2. In the **New File** dialog box, select the **General** category on the left, and then select **Code Analysis Rule Set**.
3. Select **Open**.

   The new *.ruleset* file opens in the rule set editor.

Create a custom rule set from multiple rule sets

> **NOTE**
>
> The following procedure does not apply to .NET Core projects, which don't have a **Code Analysis** property tab.

1. In **Solution Explorer**, right-click the project and then select **Properties**.

2. On the **Properties** pages, select the **Code Analysis** tab.

3. Select **<Choose multiple rule sets>** from **Run this rule set**.

3. Select **<Choose multiple rule sets>** from **Active rules**.

4. In the **Add or Remove Rule Sets** dialog box, select the rule sets you want to include in your new rule set.

5. Select **Save As**, enter a name for the *.ruleset* file, and then select **Save**.

   The new rule set is selected in the **Run this rule set** list.

6. Select **Open** to open the new rule set in the rule set editor.

Rule precedence

- If the same rule is listed two or more times in a rule set with different severities, the compiler generates an error. For example:

```
<RuleSet Name="Rules for ClassLibrary21" Description="Code analysis rules for ClassLibrary21.csproj."
ToolsVersion="15.0">
   <Rules AnalyzerId="Microsoft.Analyzers.ManagedCodeAnalysis"
RuleNamespace="Microsoft.Rules.Managed">
     <Rule Id="CA1021" Action="Warning" />
     <Rule Id="CA1021" Action="Error" />
   </Rules>
</RuleSet>
```

- If the same rule is listed two or more times in a rule set with the *same* severity, you may see the following warning in the **Error List**:

  CA0063 : Failed to load rule set file '[your].ruleset' or one of its dependent rule set files. The file does not conform to the rule set schema.

- If the rule set includes a child rule set by using an **Include** tag, and the child and parent rule sets both list

the same rule but with different severities, then the severity in the parent rule set takes precedence. For example:

```
<!-- Parent rule set -->
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="Rules for ClassLibrary21" Description="Code analysis rules for ClassLibrary21.csproj."
ToolsVersion="15.0">
   <Include Path="classlibrary_child.ruleset" Action="Default" />
   <Rules AnalyzerId="Microsoft.Analyzers.ManagedCodeAnalysis"
RuleNamespace="Microsoft.Rules.Managed">
     <Rule Id="CA1021" Action="Warning" /> <!-- Overrides CA1021 severity from child rule set -->
   </Rules>
</RuleSet>

<!-- Child rule set -->
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="Rules from child" Description="Code analysis rules from child." ToolsVersion="15.0">
   <Rules AnalyzerId="Microsoft.Analyzers.ManagedCodeAnalysis"
RuleNamespace="Microsoft.Rules.Managed">
     <Rule Id="CA1021" Action="Error" />
   </Rules>
</RuleSet>
```

Name and description

To change the display name of a rule set that's open in the editor, open the **Properties** window by selecting **View** > **Properties Window** on the menu bar. Enter the display name in the **Name** box. You can also enter a description for the rule set.

Next steps

Now that you have a rule set, the next step is to customize the rules by adding or removing rules or modifying the severity of rule violations.

Modify rules in the rule set editor

See also

- How to: Configure Code Analysis for a Managed Code Project
- Code analysis rule set reference

Use the code analysis rule set editor

1/1/2020 • 3 minutes to read • Edit Online

The code analysis rule set editor lets you specify the rules that are included in a custom rule set and set the severity of rule violations.

The following table shows the severity options:

ACTION (SEVERITY)	DESCRIPTION
Warning	Generates a warning in the **Error List** and also at build time.
Error	Generates an error in the **Error List** and also at build time.
Info	Generates a message in the **Error List**.
Hidden	The violation is not visible to the user. The IDE is notified of the violation, however.
None	The rule is suppressed. The behavior is the same as if the rule was removed from the rule set.

The editor displays the rules in a tree structure that groups the rules by a rule set field that you specify. To add or remove rules from a rule set, perform one or more of the following steps:

- Select or clear the check box of the group node to add or remove all the rules in the group. When you select a group, all rules are set to the **Warning** action.

  > **TIP**
  > You can change how rules are grouped in the **Group by** drop-down.

- Click the **Action** field of a group, and then specify the action to apply to all rules in the group.
- Select or clear the check box for an individual rule. When you select the check box for a rule, the rule is set to the Warning action.

Toolbar

You can use the toolbar of the rule set editor to group, filter, and search the data that appears in the rule set grid.

The following table describes the controls on the toolbar of the rule set editor.

TOOLBAR CONTROL	DESCRIPTION
Expand All	Shows the rules in all groups.
Collapse All	Hides the rules in all groups.
Group By	Specifies the field by which rules are grouped. Click **<None>** to show the rules without groups.

TOOLBAR CONTROL	DESCRIPTION
Column Options	Specifies the rule fields to display.
Hide rules that do not apply to the current solution	Shows or hides rules that are not of the same Target Type as the solution.
Show rules that can generate Code Analysis errors	Shows or hides rules that are assigned the Error action.
Show rules that can generate Code Analysis warnings	Shows or hides rules that are assigned the Warning action.
Show rules that are not enabled	Shows or hides rules that are assigned the None action.
Add or remove child rule sets	Adds or removes the rules in the selected rule sets.
Search rules	Searches all field values for the string that you specify.

Rule set fields

Rule set fields display information about a rule set, and can be used to sort and group the rule list. To display or hide fields, select **Column Options** on the rule set editor toolbar, and then check or clear the check boxes of the fields to show or hide.

The following table describes the fields of a rule set:

FIELD	DESCRIPTION
ID	The identifier of the rule.
Category	In addition to their membership in rule sets, code analysis rules are also grouped by category. For more information, see Code analysis warnings.
Name	The title of the rule.
Namespace	The namespace of the rule.
Target Type	Indicates whether the rule is for native, managed, or database code.
Action	The action taken when the rule is violated in a code analysis run. You can edit the **Action** field.
Source Rule Sets	The rule set that contains the rule.

Sort and filter rule sets

From the column headers of the rule set grid, you can sort and filter the rules by the values of the field.

- To sort the rule set lists, click the column header of the field by which you want to sort. If the rule sets are grouped, each group is sorted individually.
- To filter the rule sets by the value of a field, click the filter button on the column header of the field by which you want to filter. Select the check boxes of the values that you want to show, and clear the check boxes of

the values that you want to hide.

See also

- Create a custom rule set

Code metrics values

8/4/2020 • 4 minutes to read • Edit Online

The increased complexity of modern software applications also increases the difficulty of making the code reliable and maintainable. Code metrics is a set of software measures that provide developers better insight into the code they are developing. By taking advantage of code metrics, developers can understand which types and/or methods should be reworked or more thoroughly tested. Development teams can identify potential risks, understand the current state of a project, and track progress during software development.

Developers can use Visual Studio to generate code metrics data that measure the complexity and maintainability of their managed code. Code metrics data can be generated for an entire solution or a single project.

For information about how to generate code metrics data in Visual Studio, see How to: Generate code metrics data.

Software measurements

The following list shows the code metrics results that Visual Studio calculates:

- **Maintainability Index** - Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. Color coded ratings can be used to quickly identify trouble spots in your code. A green rating is between 20 and 100 and indicates that the code has good maintainability. A yellow rating is between 10 and 19 and indicates that the code is moderately maintainable. A red rating is a rating between 0 and 9 and indicates low maintainability. For more information, see the Maintainability index range and meaning blog post.
- **Cyclomatic Complexity** - Measures the structural complexity of the code. It is created by calculating the number of different code paths in the flow of the program. A program that has complex control flow requires more tests to achieve good code coverage and is less maintainable. For more information, see the Wikipedia entry for cyclomatic complexity.
- **Depth of Inheritance** - Indicates the number of different classes that inherit from one another, all the way back to the base class. Depth of Inheritance is similar to class coupling in that a change in a base class can affect any of its inherited classes. The higher this number, the deeper the inheritance and the higher the potential for base class modifications to result in a breaking change. For Depth of Inheritance, a low value is good and a high value is bad.
- **Class Coupling** - Measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration. Good software design dictates that types and methods should have high cohesion and low coupling. High coupling indicates a design that is difficult to reuse and maintain because of its many interdependencies on other types. For more information, see the Class coupling blog post.
- **Lines of Source code** - Indicates the exact number of source code lines that are present in your source file, including blank lines. This metric is available starting in Visual Studio 2019 version 16.4 and Microsoft.CodeAnalysis.Metrics (2.9.5).
- **Lines of Executable code** - Indicates the approximate number of executable code lines or operations. This is a count of number of operations in executable code. This metric is available starting in Visual Studio 2019 version 16.4 and Microsoft.CodeAnalysis.Metrics (2.9.5). The value is typically a close match to the previous metric, **Lines of Code**, which is the MSIL-instruction-based metric used in legacy mode.

- **Lines of Code** - Indicates the approximate number of lines in the code. The count is based on the IL code and is therefore not the exact number of lines in the source code file. A high count might indicate that a type or method is trying to do too much work and should be split up. It might also indicate that the type or method might be hard to maintain.

> **NOTE**
>
> The command-line version of the code metrics tool counts actual lines of code because it analyzes the source code instead of IL.

Anonymous methods

An *anonymous method* is just a method that has no name. Anonymous methods are most frequently used to pass a code block as a delegate parameter. Code metrics results for an anonymous method that's declared in a member, such as a method or accessor, are associated with the member that declares the method. They are not associated with the member that calls the method.

Generated code

Some software tools and compilers generate code that is added to a project and that the project developer either does not see or should not change. Mostly, Code Metrics ignores generated code when it calculates the metrics values. This enables the metrics values to reflect what the developer can see and change.

Code generated for Windows Forms is not ignored, because it is code that the developer can see and change.

Next steps

- How to: Generate code metrics data
- Use the Code Metrics Results window

How to: Generate code metrics data

8/4/2020 • 7 minutes to read • Edit Online

You can generate code metrics data in three ways:

- By installing FxCop analyzers and enabling the four code metrics (maintainability) rules it contains.
- By choosing the **Analyze** > **Calculate Code Metrics** menu command within Visual Studio.
- From the command line for C# and Visual Basic projects.

FxCop analyzers code metrics rules

The FxCopAnalyzers NuGet package includes several code metrics analyzer rules:

- CA1501
- CA1502
- CA1505
- CA1506

These rules are disabled by default but you can enable them from **Solution Explorer** or in a rule set file. For example, to enable rule CA1502 as a warning, your .ruleset file would contain the following entry:

```
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="Rules" Description="Rules" ToolsVersion="16.0">
  <Rules AnalyzerId="Microsoft.CodeQuality.Analyzers" RuleNamespace="Microsoft.CodeQuality.Analyzers">
    <Rule Id="CA1502" Action="Warning" />
  </Rules>
</RuleSet>
```

Configuration

You can configure the thresholds at which the code metrics rules in the FxCop analyzers package fire.

1. Create a text file. As an example, you can name it *CodeMetricsConfig.txt*.
2. Add the desired thresholds to the text file in the following format:

   ```
   CA1502: 10
   ```

   In this example, rule CA1502 is configured to fire when a method's cyclomatic complexity is greater than 10.
3. In the **Properties** window of Visual Studio, or in the project file, mark the build action of the configuration file as **AdditionalFiles**. For example:

   ```
   <ItemGroup>
     <AdditionalFiles Include="CodeMetricsConfig.txt" />
   </ItemGroup>
   ```

Calculate Code Metrics menu command

Generate code metrics for one or all of your open projects in the IDE by using the **Analyze** > **Calculate Code Metrics** menu.

Generate code metrics results for an entire solution

You can generate code metrics results for an entire solution in any of the following ways:

- From the menu bar, choose **Analyze** > **Calculate Code Metrics** > **For Solution**.
- In **Solution Explorer**, right-click the solution and then choose **Calculate Code Metrics**.
- In the **Code Metrics Results** window, choose the **Calculate Code Metrics for Solution** button.

The results are generated and the **Code Metrics Results** window is displayed. To view the results details, expand the tree in the **Hierarchy** column.

Generate code metrics results for one or more projects

1. In **Solution Explorer**, select one or more projects.
2. From the menu bar, choose **Analyze** > **Calculate Code Metrics** > **For Selected Project(s)**.

The results are generated and the **Code Metrics Results** window is displayed. To view the results details, expand the tree in the **Hierarchy**.

> **NOTE**
>
> The **Calculate Code Metrics** command does not work for .NET Core and .NET Standard projects. To calculate code metrics for a .NET Core or .NET Standard project, you can:
>
> - Calculate code metrics from the command line instead
> - Upgrade to Visual Studio 2019

Command-line code metrics

You can generate code metrics data from the command line for C# and Visual Basic projects for .NET Framework, .NET Core, and .NET Standard apps. To run code metrics from the command line, install the Microsoft.CodeAnalysis.Metrics NuGet package or build the Metrics.exe executable yourself.

Microsoft.CodeAnalysis.Metrics NuGet package

The easiest way to generate code metrics data from the command line is by installing the Microsoft.CodeAnalysis.Metrics NuGet package. After you've installed the package, run `msbuild /t:Metrics` from the directory that contains your project file. For example:

```
C:\source\repos\ClassLibrary3\ClassLibrary3>msbuild /t:Metrics
Microsoft (R) Build Engine version 16.0.360-preview+g9781d96883 for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

Build started 1/22/2019 4:29:57 PM.
Project "C:\source\repos\ClassLibrary3\ClassLibrary3\ClassLibrary3.csproj" on node 1 (Metrics target(s))
.
Metrics:
  C:\source\repos\ClassLibrary3\packages\Microsoft.CodeMetrics.2.6.4-ci\build\\..\Metrics\Metrics.exe
/project:C:\source\repos\ClassLibrary3\ClassLibrary3\ClassLibrary3.csproj /out:ClassLibrary3.Metrics.xml
  Loading ClassLibrary3.csproj...
  Computing code metrics for ClassLibrary3.csproj...
  Writing output to 'ClassLibrary3.Metrics.xml'...
  Completed Successfully.
Done Building Project "C:\source\repos\ClassLibrary3\ClassLibrary3\ClassLibrary3.csproj" (Metrics target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

You can override the output file name by specifying `/p:MetricsOutputFile=<filename>`. You can also get legacy-style code metrics data by specifying `/p:LEGACY_CODE_METRICS_MODE=true`. For example:

```
C:\source\repos\ClassLibrary3\ClassLibrary3>msbuild /t:Metrics /p:LEGACY_CODE_METRICS_MODE=true
/p:MetricsOutputFile="Legacy.xml"
Microsoft (R) Build Engine version 16.0.360-preview+g9781d96883 for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

Build started 1/22/2019 4:31:00 PM.
The "MetricsOutputFile" property is a global property, and cannot be modified.
Project "C:\source\repos\ClassLibrary3\ClassLibrary3\ClassLibrary3.csproj" on node 1 (Metrics target(s))
.
Metrics:
  C:\source\repos\ClassLibrary3\packages\Microsoft.CodeMetrics.2.6.4-
ci\build\\..\Metrics.Legacy\Metrics.Legacy.exe
/project:C:\source\repos\ClassLibrary3\ClassLibrary3\ClassLibrary3.csproj /out:Legacy.xml
  Loading ClassLibrary3.csproj...
  Computing code metrics for ClassLibrary3.csproj...
  Writing output to 'Legacy.xml'...
  Completed Successfully.
Done Building Project "C:\source\repos\ClassLibrary3\ClassLibrary3\ClassLibrary3.csproj" (Metrics target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

Code metrics output

The generated XML output takes the following format:

```xml
<?xml version="1.0" encoding="utf-8"?>
<CodeMetricsReport Version="1.0">
  <Targets>
    <Target Name="ConsoleApp20.csproj">
      <Assembly Name="ConsoleApp20, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
        <Metrics>
          <Metric Name="MaintainabilityIndex" Value="100" />
          <Metric Name="CyclomaticComplexity" Value="1" />
          <Metric Name="ClassCoupling" Value="1" />
          <Metric Name="DepthOfInheritance" Value="1" />
          <Metric Name="SourceLines" Value="11" />
          <Metric Name="ExecutableLines" Value="1" />
        </Metrics>
        <Namespaces>
          <Namespace Name="ConsoleApp20">
            <Metrics>
              <Metric Name="MaintainabilityIndex" Value="100" />
              <Metric Name="CyclomaticComplexity" Value="1" />
              <Metric Name="ClassCoupling" Value="1" />
              <Metric Name="DepthOfInheritance" Value="1" />
              <Metric Name="SourceLines" Value="11" />
              <Metric Name="ExecutableLines" Value="1" />
            </Metrics>
            <Types>
              <NamedType Name="Program">
                <Metrics>
                  <Metric Name="MaintainabilityIndex" Value="100" />
                  <Metric Name="CyclomaticComplexity" Value="1" />
                  <Metric Name="ClassCoupling" Value="1" />
                  <Metric Name="DepthOfInheritance" Value="1" />
                  <Metric Name="SourceLines" Value="7" />
                  <Metric Name="ExecutableLines" Value="1" />
                </Metrics>
                <Members>
                  <Method Name="void Program.Main(string[] args)"
File="C:\source\repos\ConsoleApp20\ConsoleApp20\Program.cs" Line="7">
                    <Metrics>
                      <Metric Name="MaintainabilityIndex" Value="100" />
                      <Metric Name="CyclomaticComplexity" Value="1" />
                      <Metric Name="ClassCoupling" Value="1" />
                      <Metric Name="SourceLines" Value="4" />
                      <Metric Name="ExecutableLines" Value="1" />
                    </Metrics>
                  </Method>
                </Members>
              </NamedType>
            </Types>
          </Namespace>
        </Namespaces>
      </Assembly>
    </Target>
  </Targets>
</CodeMetricsReport>
```

```
<?xml version="1.0" encoding="utf-8"?>
<CodeMetricsReport Version="1.0">
  <Targets>
    <Target Name="ConsoleApp20.csproj">
      <Assembly Name="ConsoleApp20, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
        <Metrics>
          <Metric Name="MaintainabilityIndex" Value="100" />
          <Metric Name="CyclomaticComplexity" Value="1" />
          <Metric Name="ClassCoupling" Value="1" />
          <Metric Name="DepthOfInheritance" Value="1" />
          <Metric Name="LinesOfCode" Value="11" />
        </Metrics>
        <Namespaces>
          <Namespace Name="ConsoleApp20">
            <Metrics>
              <Metric Name="MaintainabilityIndex" Value="100" />
              <Metric Name="CyclomaticComplexity" Value="1" />
              <Metric Name="ClassCoupling" Value="1" />
              <Metric Name="DepthOfInheritance" Value="1" />
              <Metric Name="LinesOfCode" Value="11" />
            </Metrics>
            <Types>
              <NamedType Name="Program">
                <Metrics>
                  <Metric Name="MaintainabilityIndex" Value="100" />
                  <Metric Name="CyclomaticComplexity" Value="1" />
                  <Metric Name="ClassCoupling" Value="1" />
                  <Metric Name="DepthOfInheritance" Value="1" />
                  <Metric Name="LinesOfCode" Value="7" />
                </Metrics>
                <Members>
                  <Method Name="void Program.Main(string[] args)"
File="C:\source\repos\ConsoleApp20\ConsoleApp20\Program.cs" Line="7">
                    <Metrics>
                      <Metric Name="MaintainabilityIndex" Value="100" />
                      <Metric Name="CyclomaticComplexity" Value="1" />
                      <Metric Name="ClassCoupling" Value="1" />
                      <Metric Name="LinesOfCode" Value="4" />
                    </Metrics>
                  </Method>
                </Members>
              </NamedType>
            </Types>
          </Namespace>
        </Namespaces>
      </Assembly>
    </Target>
  </Targets>
</CodeMetricsReport>
```

Metrics.exe

If you don't want to install the NuGet package, you can generate and use the *Metrics.exe* executable directly. To generate the *Metrics.exe* executable:

1. Clone the dotnet/roslyn-analyzers repo.
2. Open Developer Command Prompt for Visual Studio as an administrator.
3. From the root of the **roslyn-analyzers** repo, execute the following command: `Restore.cmd`
4. Change directory to *src\Tools*.
5. Execute the following command to build the **Metrics.csproj** project:

```
msbuild /m /v:m /p:Configuration=Release Metrics.csproj
```

An executable named *Metrics.exe* is generated in the *artifacts\bin* directory under the repo root.

Metrics.exe usage

To run *Metrics.exe*, supply a project or solution and an output XML file as arguments. For example:

```
C:\>Metrics.exe /project:ConsoleApp20.csproj /out:report.xml
Loading ConsoleApp20.csproj...
Computing code metrics for ConsoleApp20.csproj...
Writing output to 'report.xml'...
Completed Successfully.
```

Legacy mode

You can choose to build *Metrics.exe* in *legacy mode*. The legacy mode version of the tool generates metric values that are closer to what older versions of the tool generated. Additionally, in legacy mode, *Metrics.exe* generates code metrics for the same set of method types that previous versions of the tool generated code metrics for. For example, it doesn't generate code metrics data for field and property initializers. Legacy mode is useful for backwards compatibility or if you have code check-in gates based on code metrics numbers. The command to build *Metrics.exe* in legacy mode is:

```
msbuild /m /v:m /t:rebuild /p:LEGACY_CODE_METRICS_MODE=true Metrics.csproj
```

For more information, see Enable generating code metrics in legacy mode.

Previous versions

Visual Studio 2015 included a command-line code metrics tool that was also called *Metrics.exe*. This previous version of the tool did a binary analysis, that is, an assembly-based analysis. The newer version of the *Metrics.exe* tool analyzes source code instead. Because the newer *Metrics.exe* tool is source code-based, command-line code metrics results may be different to those generated by the Visual Studio IDE and by previous versions of *Metrics.exe*. Starting in Visual Studio 2019, the Visual Studio IDE analyzes source code like the command-line tool and the results should be the same.

Visual Studio 2015 included a command-line code metrics tool that was also called *Metrics.exe*. This previous version of the tool did a binary analysis, that is, an assembly-based analysis. The new *Metrics.exe* tool analyzes source code instead. Because the new *Metrics.exe* tool is source code-based, command-line code metrics results are different to those generated by the Visual Studio IDE and by previous versions of *Metrics.exe*.

The new command-line code metrics tool computes metrics even in the presence of source code errors, as long as the solution and project can be loaded.

Metric value differences

Starting in Visual Studio 2019 version 16.4 and Microsoft.CodeAnalysis.Metics (2.9.5), `SourceLines` and `ExecutableLines` replace the previous `LinesOfCode` metric. For descriptions of the new metrics, see Code metrics values. The `LinesOfCode` metric is available in legacy mode.

The `LinesOfCode` metric is more accurate and reliable in the new command-line code metrics tool. It's independent of any codegen differences and doesn't change when the toolset or runtime changes. The new tool counts actual lines of code, including blank lines and comments.

Other metrics such as `CyclomaticComplexity` and `MaintainabilityIndex` use the same formulas as previous versions of *Metrics.exe*, but the new tool counts the number of `IOperations` (logical source instructions) instead of intermediate language (IL) instructions. The numbers will be slightly different to those generated by the Visual Studio IDE and by previous versions of *Metrics.exe*.

See also

- Use the Code Metrics Results window
- Code metrics values

Use the Code Metrics Results window

1/1/2020 • 3 minutes to read • Edit Online

The **Code Metrics Results** window displays the data that is generated by the code metrics analysis. For more information about code metrics data values, see Code metrics values.

Display code metrics results

The **Code Metrics Results** window is displayed automatically when you generate code metrics results. You can also display the window at any time.

You can display the Code Metrics Results window using one of the following menu sequences:

- On the **Analyze** menu, choose **Windows** > **Code Metrics Results**.
- On the **View** menu, choose **Other Windows** > **Code Metrics Results**.

The **Code Metrics Results** window opens, even if it contains no results.

To view code metrics details

If code metrics results have been generated, expand the tree in the **Hierarchy** column.

Filter code metrics results

You can filter the results that are displayed in the **Code Metrics Results** window by using the toolbar at the top. For example, you might want to see only the results that have a maintainability index below 65.

The **Filter** drop-down box contains the names of the results columns. When a filter is defined, it is added to the bottom of the list together with an indentation. The list can contain the last 10 filters that were defined.

To filter the code metrics results

1. From the **Filter** list, select the column name.
2. In **Min**, type the minimum value to be displayed.
3. In **Max**, type the maximum value to be displayed.
4. Click the **Apply Filter** button.
5. To see the result details, expand the hierarchy tree.

Add, remove, and rearrange data columns

You can add or remove results columns from the **Code Metrics Results** window. In addition, you can rearrange results columns so that they appear in the order that you want.

Add or remove a column

1. Click the **Add/Remove Columns** button, or right-click any column heading and then click **Add/Remove Columns**.
2. In the **Add/Remove Columns** dialog box, select or clear the check box for the column that you want to add or remove, and then choose **OK**.

Rearrange columns

1. Click the **Add/Remove Columns** button.
2. In the **Add/Remove Columns** dialog box, select the column that you want to move and then choose either the up arrow or the down arrow.
3. When the column is positioned where you want it, choose **OK**.

Copy data to the clipboard or Excel

You can select and copy a selected row of code metrics data to the clipboard as a text string that contains one line for the name and value of each data column. You can also click **Open Selection in Microsoft Excel** to export all of the code metrics results to an Excel spreadsheet.

Create a work item based on code metric results

You can create an Azure Boards work item that is based on results in the **Code Metric Results** window. When the work item is created, Visual Studio automatically enters a title in the **Title** field and code metrics data under the **History** tab.

For more information about Azure Boards work items, see Work items.

To create a work item based on a result

1. Right-click the result.
2. Point to **Create Work Item**, and then click the type of work item you want to create (**Bug**, **Task**, and so forth).
3. Complete the work item form by filling in all required fields.
4. On the **File** menu, click **Save All** to save the work item.

To create a bug based on a result

1. Click the result to select it.
2. Click the **Create Work Item** button.
3. Complete the work item form by filling in all required fields.
4. On the **File** menu, click **Save All** to save the work item.

See also

- Code metrics values
- How to: Generate code metrics data

Code analysis rule set reference

1/1/2020 • 2 minutes to read • Edit Online

When you configure legacy analysis for managed code projects in Visual Studio, you can choose from a list of built-in *rule sets*. Some rules are included in more than one of the built-in rule sets, for example, the Basic Correctness Rules rule set includes rules that are in the Managed Recommended Rules rule set.

> **NOTE**
>
> The rule sets in this section pertain to legacy analysis. For information about rule sets available for code analyzer packages, see Use rule sets with code analyzers.

You can either use one of these built-in rule sets, or you can customize a rule set to fit your project requirements. If you include multiple rule sets that contain the same rule in a custom rule set, that rule only appears once in the custom rule set.

The topics in this section describe the built-in rule sets and the rules (or warnings) they contain.

RULE SET	INCLUDED RULES
All Rules	Contains all available managed and C++ rules
Basic Correctness Rules	Includes Managed Recommended Rules plus rules for logic errors and framework usage
Extended Correctness Rules	Includes Basic Correctness Rules (which includes Managed Recommended Rules) plus more rules for logic errors and framework usage
Basic Design Guideline Rules	Includes Managed Recommended Rules plus rules for ensuring code is easy to read, understand, and maintain
Extended Design Guidelines Rules	Includes Basic Design Guideline Rules (which includes Managed Recommended Rules) plus more maintainability rules that focus on naming
Globalization Rules	Includes rules for globalization problems
Managed Minimum Rules	Includes four rules for critical managed code problems
Managed Recommended Rules	Includes Managed Minimum Rules plus more rules for critical managed code problems
Mixed Minimum Rules	Includes rules for critical problems in C++ code for CLR
Mixed Recommended Rules	Includes Mixed Minimum Rules plus more rules for critical problems in C++ code for CLR
Native Minimum Rules	Includes rules for critical problems in native code

RULE SET	INCLUDED RULES
Native Recommended Rules	Includes Native Minimum Rules plus more rules for critical problems in native code
Security Rules	Includes rules for finding security vulnerabilities

All Rules rule set

2/19/2020 • 2 minutes to read • Edit Online

The All Rules rule set contains all of the rules for both native and managed code. The rule set includes all the rules that are described in the following topics:

- Code Analysis for C/C++ Warnings
- Code Analysis for Managed Code Warnings

Basic Correctness Rules rule set for managed code

1/1/2020 • 3 minutes to read • Edit Online

The Basic Correctness Rules rule set focuses on logic errors and common mistakes in the usage of framework APIs. The Basic Correctness Rules include the rules in the Managed Recommended Rules rule set.

The following table describes all the rules in the Microsoft Basic Correctness Rules rule set.

RULE	DESCRIPTION
CA1001	Types that own disposable fields should be disposable
CA1009	Declare event handlers correctly
CA1016	Mark assemblies with AssemblyVersionAttribute
CA1033	Interface methods should be callable by child types
CA1049	Types that own native resources should be disposable
CA1060	Move P/Invokes to NativeMethods class
CA1061	Do not hide base class methods
CA1063	Implement IDisposable correctly
CA1065	Do not raise exceptions in unexpected locations
CA1301	Avoid duplicate accelerators
CA1400	P/Invoke entry points should exist
CA1401	P/Invokes should not be visible
CA1403	Auto layout types should not be COM visible
CA1404	Call GetLastError immediately after P/Invoke
CA1405	COM visible type base types should be COM visible
CA1410	COM registration methods should be matched
CA1415	Declare P/Invokes correctly
CA1821	Remove empty finalizers
CA1900	Value type fields should be portable
CA1901	P/Invoke declarations should be portable

RULE	DESCRIPTION
CA2002	Do not lock on objects with weak identity
CA2100	Review SQL queries for security vulnerabilities
CA2101	Specify marshaling for P/Invoke string arguments
CA2108	Review declarative security on value types
CA2111	Pointers should not be visible
CA2112	Secured types should not expose fields
CA2114	Method security should be a superset of type
CA2116	APTCA methods should only call APTCA methods
CA2117	APTCA types should only extend APTCA base types
CA2122	Do not indirectly expose methods with link demands
CA2123	Override link demands should be identical to base
CA2124	Wrap vulnerable finally clauses in outer try
CA2126	Type link demands require inheritance demands
CA2131	Security critical types may not participate in type equivalence
CA2132	Default constructors must be at least as critical as base type default constructors
CA2133	Delegates must bind to methods with consistent transparency
CA2134	Methods must keep consistent transparency when overriding base methods
CA2137	Transparent methods must contain only verifiable IL
CA2138	Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute
CA2140	Transparent code must not reference security critical items
CA2141	Transparent methods must not satisfy LinkDemands
CA2146	Types must be at least as critical as their base types and interfaces
CA2147	Transparent methods may not use security asserts
CA2149	Transparent methods must not call into native code

RULE	DESCRIPTION
CA2200	Rethrow to preserve stack details
CA2202	Do not dispose objects multiple times
CA2207	Initialize value type static fields inline
CA2212	Do not mark serviced components with WebMethod
CA2213	Disposable fields should be disposed
CA2214	Do not call overridable methods in constructors
CA2216	Disposable types should declare finalizer
CA2220	Finalizers should call base class finalizer
CA2229	Implement serialization constructors
CA2231	Overload operator equals on overriding ValueType.Equals
CA2232	Mark Windows Forms entry points with STAThread
CA2235	Mark all non-serializable fields
CA2236	Call base class methods on ISerializable types
CA2237	Mark ISerializable types with SerializableAttribute
CA2238	Implement serialization methods correctly
CA2240	Implement ISerializable correctly
CA2241	Provide correct arguments to formatting methods
CA2242	Test for NaN correctly
CA1008	Enums should have zero value
CA1013	Overload operator equals on overloading add and subtract
CA1303	Do not pass literals as localized parameters
CA1308	Normalize strings to uppercase
CA1806	Do not ignore method results
CA1816	Call GC.SuppressFinalize correctly
CA1819	Properties should not return arrays

RULE	DESCRIPTION
CA1820	Test for empty strings using string length
CA1903	Use only API from targeted framework
CA2004	Remove calls to GC.KeepAlive
CA2006	Use SafeHandle to encapsulate native resources
CA2102	Catch non-CLSCompliant exceptions in general handlers
CA2104	Do not declare read only mutable reference types
CA2105	Array fields should not be read only
CA2106	Secure asserts
CA2115	Call GC.KeepAlive when using native resources
CA2119	Seal methods that satisfy private interfaces
CA2120	Secure serialization constructors
CA2121	Static constructors should be private
CA2130	Security critical constants should be transparent
CA2205	Use managed equivalents of Win32 API
CA2215	Dispose methods should call base class dispose
CA2221	Finalizers should be protected
CA2222	Do not decrease inherited member visibility
CA2223	Members should differ by more than return type
CA2224	Override equals on overloading operator equals
CA2226	Operators should have symmetrical overloads
CA2227	Collection properties should be read only
CA2239	Provide deserialization methods for optional fields

Basic Design Guideline Rules rule set for managed code

1/1/2020 • 5 minutes to read • Edit Online

You can use the Microsoft Basic Design Guideline Rules rule set to focus on making your code easier to understand and use. You should include this rule set if your project includes library code or if you want to enforce best practices for code that is easy to maintain.

The Basic Design Guideline Rules include all the rules in the Managed Recommended Rules rule set.

The following table describes all of the rules in the Microsoft Basic Design Guideline Rules rule set.

RULE	DESCRIPTION
CA1000	Do not declare static members on generic types
CA1001	Types that own disposable fields should be disposable
CA1002	Do not expose generic lists
CA1003	Use generic event handler instances
CA1004	Generic methods should provide type parameter
CA1005	Avoid excessive parameters on generic types
CA1006	Do not nest generic types in member signatures
CA1007	Use generics where appropriate
CA1008	Enums should have zero value
CA1009	Declare event handlers correctly
CA1010	Collections should implement generic interface
CA1011	Consider passing base types as parameters
CA1012	Abstract types should not have constructors
CA1013	Overload operator equals on overloading add and subtract
CA1014	Mark assemblies with CLSCompliantAttribute
CA1016	Mark assemblies with AssemblyVersionAttribute
CA1017	Mark assemblies with ComVisibleAttribute
CA1018	Mark attributes with AttributeUsageAttribute

RULE	DESCRIPTION
CA1019	Define accessors for attribute arguments
CA1023	Indexers should not be multidimensional
CA1024	Use properties where appropriate
CA1025	Replace repetitive arguments with params array
CA1026	Default parameters should not be used
CA1027	Mark enums with FlagsAttribute
CA1028	Enum storage should be Int32
CA1030	Use events where appropriate
CA1031	Do not catch general exception types
CA1032	Implement standard exception constructors
CA1033	Interface methods should be callable by child types
CA1034	Nested types should not be visible
CA1035	ICollection implementations have strongly typed members
CA1036	Override methods on comparable types
CA1038	Enumerators should be strongly typed
CA1039	Lists are strongly typed
CA1041	Provide ObsoleteAttribute message
CA1043	Use integral or string argument for indexers
CA1044	Properties should not be write only
CA1046	Do not overload operator equals on reference types
CA1047	Do not declare protected members in sealed types
CA1048	Do not declare virtual members in sealed types
CA1049	Types that own native resources should be disposable
CA1050	Declare types in namespaces
CA1051	Do not declare visible instance fields

RULE	DESCRIPTION
CA1052	Static holder types should be sealed
CA1053	Static holder types should not have constructors
CA1054	URI parameters should not be strings
CA1055	URI return values should not be strings
CA1056	URI properties should not be strings
CA1057	String URI overloads call System.Uri overloads
CA1058	Types should not extend certain base types
CA1059	Members should not expose certain concrete types
CA1060	Move P/Invokes to NativeMethods class
CA1061	Do not hide base class methods
CA1063	Implement IDisposable correctly
CA1064	Exceptions should be public
CA1065	Do not raise exceptions in unexpected locations
CA1301	Avoid duplicate accelerators
CA1400	P/Invoke entry points should exist
CA1401	P/Invokes should not be visible
CA1403	Auto layout types should not be COM visible
CA1404	Call GetLastError immediately after P/Invoke
CA1405	COM visible type base types should be COM visible
CA1410	COM registration methods should be matched
CA1415	Declare P/Invokes correctly
CA1500	Variable names should not match field names
CA1502	Avoid excessive complexity
CA1708	Identifiers should differ by more than case
CA1716	Identifiers should not match keywords

RULE	DESCRIPTION
CA1801	Review unused parameters
CA1804	Remove unused locals
CA1809	Avoid excessive locals
CA1810	Initialize reference type static fields inline
CA1811	Avoid uncalled private code
CA1812	Avoid uninstantiated internal classes
CA1813	Avoid unsealed attributes
CA1814	Prefer jagged arrays over multidimensional
CA1815	Override equals and operator equals on value types
CA1819	Properties should not return arrays
CA1820	Test for empty strings using string length
CA1821	Remove empty finalizers
CA1822	Mark members as static
CA1823	Avoid unused private fields
CA1900	Value type fields should be portable
CA1901	P/Invoke declarations should be portable
CA2002	Do not lock on objects with weak identity
CA2100	Review SQL queries for security vulnerabilities
CA2101	Specify marshaling for P/Invoke string arguments
CA2108	Review declarative security on value types
CA2111	Pointers should not be visible
CA2112	Secured types should not expose fields
CA2114	Method security should be a superset of type
CA2116	APTCA methods should only call APTCA methods
CA2117	APTCA types should only extend APTCA base types

RULE	DESCRIPTION
CA2122	Do not indirectly expose methods with link demands
CA2123	Override link demands should be identical to base
CA2124	Wrap vulnerable finally clauses in outer try
CA2126	Type link demands require inheritance demands
CA2131	Security critical types may not participate in type equivalence
CA2132	Default constructors must be at least as critical as base type default constructors
CA2133	Delegates must bind to methods with consistent transparency
CA2134	Methods must keep consistent transparency when overriding base methods
CA2137	Transparent methods must contain only verifiable IL
CA2138	Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute
CA2140	Transparent code must not reference security critical items
CA2141	Transparent methods must not satisfy LinkDemands
CA2146	Types must be at least as critical as their base types and interfaces
CA2147	Transparent methods may not use security asserts
CA2149	Transparent methods must not call into native code
CA2200	Rethrow to preserve stack details
CA2201	Do not raise reserved exception types
CA2202	Do not dispose objects multiple times
CA2205	Use managed equivalents of Win32 API
CA2207	Initialize value type static fields inline
CA2208	Instantiate argument exceptions correctly
CA2211	Non-constant fields should not be visible
CA2212	Do not mark serviced components with WebMethod
CA2213	Disposable fields should be disposed

RULE	DESCRIPTION
CA2214	Do not call overridable methods in constructors
CA2216	Disposable types should declare finalizer
CA2217	Do not mark enums with FlagsAttribute
CA2219	Do not raise exceptions in exception clauses
CA2220	Finalizers should call base class finalizer
CA2221	Finalizers should be protected
CA2222	Do not decrease inherited member visibility
CA2223	Members should differ by more than return type
CA2224	Override equals on overloading operator equals
CA2225	Operator overloads have named alternates
CA2226	Operators should have symmetrical overloads
CA2227	Collection properties should be read only
CA2229	Implement serialization constructors
CA2230	Use params for variable arguments
CA2231	Overload operator equals on overriding ValueType.Equals
CA2232	Mark Windows Forms entry points with STAThread
CA2234	Pass System.Uri objects instead of strings
CA2235	Mark all non-serializable fields
CA2236	Call base class methods on ISerializable types
CA2237	Mark ISerializable types with SerializableAttribute
CA2238	Implement serialization methods correctly
CA2239	Provide deserialization methods for optional fields
CA2240	Implement ISerializable correctly
CA2241	Provide correct arguments to formatting methods
CA2242	Test for NaN correctly

Extended Correctness Rules rule set for managed code

1/1/2020 • 5 minutes to read • Edit Online

The Microsoft Extended Correctness Rules rule set maximizes the logic and framework usage errors that are reported by code analysis. Extra emphasis is placed on specific scenarios such as COM interoperability and mobile applications. You should consider including this rule set if one of these scenarios applies to your project or to find additional problems in your project.

The Microsoft Extended Correctness Rules rule set includes the rules that are in the Basic Correctness Rules rule set, which contains the rules that are in the Managed Recommended Rules rule set.

The following table describes all of the rules in the Microsoft Extended Correctness Rules rule set.

RULE	DESCRIPTION
CA1001	Types that own disposable fields should be disposable
CA1009	Declare event handlers correctly
CA1016	Mark assemblies with AssemblyVersionAttribute
CA1033	Interface methods should be callable by child types
CA1049	Types that own native resources should be disposable
CA1060	Move P/Invokes to NativeMethods class
CA1061	Do not hide base class methods
CA1063	Implement IDisposable correctly
CA1065	Do not raise exceptions in unexpected locations
CA1301	Avoid duplicate accelerators
CA1400	P/Invoke entry points should exist
CA1401	P/Invokes should not be visible
CA1403	Auto layout types should not be COM visible
CA1404	Call GetLastError immediately after P/Invoke
CA1405	COM visible type base types should be COM visible
CA1410	COM registration methods should be matched
CA1415	Declare P/Invokes correctly

RULE	DESCRIPTION
CA1821	Remove empty finalizers
CA1900	Value type fields should be portable
CA1901	P/Invoke declarations should be portable
CA2002	Do not lock on objects with weak identity
CA2100	Review SQL queries for security vulnerabilities
CA2101	Specify marshaling for P/Invoke string arguments
CA2108	Review declarative security on value types
CA2111	Pointers should not be visible
CA2112	Secured types should not expose fields
CA2114	Method security should be a superset of type
CA2116	APTCA methods should only call APTCA methods
CA2117	APTCA types should only extend APTCA base types
CA2122	Do not indirectly expose methods with link demands
CA2123	Override link demands should be identical to base
CA2124	Wrap vulnerable finally clauses in outer try
CA2126	Type link demands require inheritance demands
CA2131	Security critical types may not participate in type equivalence
CA2132	Default constructors must be at least as critical as base type default constructors
CA2133	Delegates must bind to methods with consistent transparency
CA2134	Methods must keep consistent transparency when overriding base methods
CA2137	Transparent methods must contain only verifiable IL
CA2138	Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute
CA2140	Transparent code must not reference security critical items
CA2141	Transparent methods must not satisfy LinkDemands

RULE	DESCRIPTION
CA2146	Types must be at least as critical as their base types and interfaces
CA2147	Transparent methods may not use security asserts
CA2149	Transparent methods must not call into native code
CA2200	Rethrow to preserve stack details
CA2202	Do not dispose objects multiple times
CA2207	Initialize value type static fields inline
CA2212	Do not mark serviced components with WebMethod
CA2213	Disposable fields should be disposed
CA2214	Do not call overridable methods in constructors
CA2216	Disposable types should declare finalizer
CA2220	Finalizers should call base class finalizer
CA2229	Implement serialization constructors
CA2231	Overload operator equals on overriding ValueType.Equals
CA2232	Mark Windows Forms entry points with STAThread
CA2235	Mark all non-serializable fields
CA2236	Call base class methods on ISerializable types
CA2237	Mark ISerializable types with SerializableAttribute
CA2238	Implement serialization methods correctly
CA2240	Implement ISerializable correctly
CA2241	Provide correct arguments to formatting methods
CA2242	Test for NaN correctly
CA1008	Enums should have zero value
CA1013	Overload operator equals on overloading add and subtract
CA1303	Do not pass literals as localized parameters
CA1308	Normalize strings to uppercase

RULE	DESCRIPTION
CA1806	Do not ignore method results
CA1816	Call GC.SuppressFinalize correctly
CA1819	Properties should not return arrays
CA1820	Test for empty strings using string length
CA1903	Use only API from targeted framework
CA2004	Remove calls to GC.KeepAlive
CA2006	Use SafeHandle to encapsulate native resources
CA2102	Catch non-CLSCompliant exceptions in general handlers
CA2104	Do not declare read only mutable reference types
CA2105	Array fields should not be read only
CA2106	Secure asserts
CA2115	Call GC.KeepAlive when using native resources
CA2119	Seal methods that satisfy private interfaces
CA2120	Secure serialization constructors
CA2121	Static constructors should be private
CA2130	Security critical constants should be transparent
CA2205	Use managed equivalents of Win32 API
CA2215	Dispose methods should call base class dispose
CA2221	Finalizers should be protected
CA2222	Do not decrease inherited member visibility
CA2223	Members should differ by more than return type
CA2224	Override equals on overloading operator equals
CA2226	Operators should have symmetrical overloads
CA2227	Collection properties should be read only
CA2239	Provide deserialization methods for optional fields

RULE	DESCRIPTION
CA1032	Implement standard exception constructors
CA1054	URI parameters should not be strings
CA1055	URI return values should not be strings
CA1056	URI properties should not be strings
CA1057	String URI overloads call System.Uri overloads
CA1402	Avoid overloads in COM visible interfaces
CA1406	Avoid Int64 arguments for Visual Basic 6 clients
CA1407	Avoid static members in COM visible types
CA1408	Do not use AutoDual ClassInterfaceType
CA1409	Com visible types should be creatable
CA1411	COM registration methods should not be visible
CA1412	Mark ComSource Interfaces as IDispatch
CA1413	Avoid non-public fields in COM visible value types
CA1414	Mark boolean P/Invoke arguments with MarshalAs
CA1600	Do not use idle process priority
CA1601	Do not use timers that prevent power state changes
CA1824	Mark assemblies with NeutralResourcesLanguageAttribute
CA2001	Avoid calling problematic methods
CA2003	Do not treat fibers as threads
CA2135	Level 2 assemblies should not contain LinkDemands
CA2136	Members should not have conflicting transparency annotations
CA2139	Transparent methods may not use the HandleProcessCorruptingExceptions attribute
CA2142	Transparent code should not be protected with LinkDemands
CA2143	Transparent methods should not use security demands

RULE	DESCRIPTION
CA2144	Transparent code should not load assemblies from byte arrays
CA2145	Transparent methods should not be decorated with the SuppressUnmanagedCodeSecurityAttribute
CA2204	Literals should be spelled correctly
CA2211	Non-constant fields should not be visible
CA2217	Do not mark enums with FlagsAttribute
CA2218	Override GetHashCode on overriding Equals
CA2219	Do not raise exceptions in exception clauses
CA2225	Operator overloads have named alternates
CA2228	Do not ship unreleased resource formats
CA2230	Use params for variable arguments
CA2233	Operations should not overflow
CA2234	Pass System.Uri objects instead of strings
CA2243	Attribute string literals should parse correctly

Extended Design Guidelines Rules rule set for managed code

1/1/2020 • 6 minutes to read • Edit Online

The Microsoft Extended Design Guideline Rules rule set expands on the basic design guideline rules to maximize the usability and maintainability issues that are reported. Extra emphasis is placed on naming guidelines. You should consider including this rule set if your project includes library code or if you want to enforce the highest standards for writing code that is easy to maintain.

The Extended Design Guideline Rules include all rules in the Basic Design Guideline Rules rule set, which includes the rules in the Managed Recommended Rules rule set.

The following table describes all the rules in the Microsoft Extended Design Guideline Rules rule set.

RULE	DESCRIPTION
CA1001	Types that own disposable fields should be disposable
CA1009	Declare event handlers correctly
CA1016	Mark assemblies with AssemblyVersionAttribute
CA1033	Interface methods should be callable by child types
CA1049	Types that own native resources should be disposable
CA1060	Move P/Invokes to NativeMethods class
CA1061	Do not hide base class methods
CA1063	Implement IDisposable correctly
CA1065	Do not raise exceptions in unexpected locations
CA1301	Avoid duplicate accelerators
CA1400	P/Invoke entry points should exist
CA1401	P/Invokes should not be visible
CA1403	Auto layout types should not be COM visible
CA1404	Call GetLastError immediately after P/Invoke
CA1405	COM visible type base types should be COM visible
CA1410	COM registration methods should be matched
CA1415	Declare P/Invokes correctly

RULE	DESCRIPTION
CA1821	Remove empty finalizers
CA1900	Value type fields should be portable
CA1901	P/Invoke declarations should be portable
CA2002	Do not lock on objects with weak identity
CA2100	Review SQL queries for security vulnerabilities
CA2101	Specify marshaling for P/Invoke string arguments
CA2108	Review declarative security on value types
CA2111	Pointers should not be visible
CA2112	Secured types should not expose fields
CA2114	Method security should be a superset of type
CA2116	APTCA methods should only call APTCA methods
CA2117	APTCA types should only extend APTCA base types
CA2122	Do not indirectly expose methods with link demands
CA2123	Override link demands should be identical to base
CA2124	Wrap vulnerable finally clauses in outer try
CA2126	Type link demands require inheritance demands
CA2131	Security critical types may not participate in type equivalence
CA2132	Default constructors must be at least as critical as base type default constructors
CA2133	Delegates must bind to methods with consistent transparency
CA2134	Methods must keep consistent transparency when overriding base methods
CA2137	Transparent methods must contain only verifiable IL
CA2138	Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute
CA2140	Transparent code must not reference security critical items
CA2141	Transparent methods must not satisfy LinkDemands

RULE	DESCRIPTION
CA2146	Types must be at least as critical as their base types and interfaces
CA2147	Transparent methods may not use security asserts
CA2149	Transparent methods must not call into native code
CA2200	Rethrow to preserve stack details
CA2202	Do not dispose objects multiple times
CA2207	Initialize value type static fields inline
CA2212	Do not mark serviced components with WebMethod
CA2213	Disposable fields should be disposed
CA2214	Do not call overridable methods in constructors
CA2216	Disposable types should declare finalizer
CA2220	Finalizers should call base class finalizer
CA2229	Implement serialization constructors
CA2231	Overload operator equals on overriding ValueType.Equals
CA2232	Mark Windows Forms entry points with STAThread
CA2235	Mark all non-serializable fields
CA2236	Call base class methods on ISerializable types
CA2237	Mark ISerializable types with SerializableAttribute
CA2238	Implement serialization methods correctly
CA2240	Implement ISerializable correctly
CA2241	Provide correct arguments to formatting methods
CA2242	Test for NaN correctly
CA1000	Do not declare static members on generic types
CA1002	Do not expose generic lists
CA1003	Use generic event handler instances
CA1004	Generic methods should provide type parameter

RULE	DESCRIPTION
CA1005	Avoid excessive parameters on generic types
CA1006	Do not nest generic types in member signatures
CA1007	Use generics where appropriate
CA1008	Enums should have zero value
CA1010	Collections should implement generic interface
CA1011	Consider passing base types as parameters
CA1012	Abstract types should not have constructors
CA1013	Overload operator equals on overloading add and subtract
CA1014	Mark assemblies with CLSCompliantAttribute
CA1017	Mark assemblies with ComVisibleAttribute
CA1018	Mark attributes with AttributeUsageAttribute
CA1019	Define accessors for attribute arguments
CA1023	Indexers should not be multidimensional
CA1024	Use properties where appropriate
CA1025	Replace repetitive arguments with params array
CA1026	Default parameters should not be used
CA1027	Mark enums with FlagsAttribute
CA1028	Enum storage should be Int32
CA1030	Use events where appropriate
CA1031	Do not catch general exception types
CA1032	Implement standard exception constructors
CA1034	Nested types should not be visible
CA1035	ICollection implementations have strongly typed members
CA1036	Override methods on comparable types
CA1038	Enumerators should be strongly typed

RULE	DESCRIPTION
CA1039	Lists are strongly typed
CA1041	Provide ObsoleteAttribute message
CA1043	Use integral or string argument for indexers
CA1044	Properties should not be write only
CA1046	Do not overload operator equals on reference types
CA1047	Do not declare protected members in sealed types
CA1048	Do not declare virtual members in sealed types
CA1050	Declare types in namespaces
CA1051	Do not declare visible instance fields
CA1052	Static holder types should be sealed
CA1053	Static holder types should not have constructors
CA1054	URI parameters should not be strings
CA1055	URI return values should not be strings
CA1056	URI properties should not be strings
CA1057	String URI overloads call System.Uri overloads
CA1058	Types should not extend certain base types
CA1059	Members should not expose certain concrete types
CA1064	Exceptions should be public
CA1500	Variable names should not match field names
CA1502	Avoid excessive complexity
CA1708	Identifiers should differ by more than case
CA1716	Identifiers should not match keywords
CA1801	Review unused parameters
CA1804	Remove unused locals
CA1809	Avoid excessive locals

RULE	DESCRIPTION
CA1810	Initialize reference type static fields inline
CA1811	Avoid uncalled private code
CA1812	Avoid uninstantiated internal classes
CA1813	Avoid unsealed attributes
CA1814	Prefer jagged arrays over multidimensional
CA1815	Override equals and operator equals on value types
CA1819	Properties should not return arrays
CA1820	Test for empty strings using string length
CA1822	Mark members as static
CA1823	Avoid unused private fields
CA2201	Do not raise reserved exception types
CA2205	Use managed equivalents of Win32 API
CA2208	Instantiate argument exceptions correctly
CA2211	Non-constant fields should not be visible
CA2217	Do not mark enums with FlagsAttribute
CA2219	Do not raise exceptions in exception clauses
CA2221	Finalizers should be protected
CA2222	Do not decrease inherited member visibility
CA2223	Members should differ by more than return type
CA2224	Override equals on overloading operator equals
CA2225	Operator overloads have named alternates
CA2226	Operators should have symmetrical overloads
CA2227	Collection properties should be read only
CA2230	Use params for variable arguments
CA2234	Pass System.Uri objects instead of strings

RULE	DESCRIPTION
CA2239	Provide deserialization methods for optional fields
CA1020	Avoid namespaces with few types
CA1021	Avoid out parameters
CA1040	Avoid empty interfaces
CA1045	Do not pass types by reference
CA1062	Validate arguments of public methods
CA1501	Avoid excessive inheritance
CA1504	Review misleading field names
CA1505	Avoid unmaintainable code
CA1506	Avoid excessive class coupling
CA1700	Do not name enum values 'Reserved'
CA1701	Resource string compound words should be cased correctly
CA1702	Compound words should be cased correctly
CA1703	Resource strings should be spelled correctly
CA1704	Identifiers should be spelled correctly
CA1707	Identifiers should not contain underscores
CA1709	Identifiers should be cased correctly
CA1710	Identifiers should have correct suffix
CA1711	Identifiers should not have incorrect suffix
CA1712	Do not prefix enum values with type name
CA1713	Events should not have before or after prefix
CA1714	Flags enums should have plural names
CA1715	Identifiers should have correct prefix
CA1717	Only FlagsAttribute enums should have plural names
CA1719	Parameter names should not match member names

RULE	DESCRIPTION
CA1720	Identifiers should not contain type names
CA1721	Property names should not match get methods
CA1722	Identifiers should not have incorrect prefix
CA1724	Type Names Should Not Match Namespaces
CA1725	Parameter names should match base declaration
CA1726	Use preferred terms
CA2204	Literals should be spelled correctly

Globalization Rules rule set for managed code

1/1/2020 • 2 minutes to read • Edit Online

Use the Microsoft Globalization Rules rule set to focus on problems that might prevent data in your application from appearing correctly in different languages, locales, and cultures. You should include this rule set if your application is localized, globalized, or both.

RULE	DESCRIPTION
CA1300	Specify MessageBoxOptions
CA1301	Avoid duplicate accelerators
CA1302	Do not hardcode locale-specific strings
CA1303	Do not pass literals as localized parameters
CA1304	Specify CultureInfo
CA1305	Specify IFormatProvider
CA1306	Set locale for data types
CA1307	Specify StringComparison
CA1308	Normalize strings to uppercase
CA1309	Use ordinal StringComparison
CA2101	Specify marshaling for P/Invoke string arguments

Managed Minimum Rules rule set for managed code

1/1/2020 • 2 minutes to read • Edit Online

The Managed Minimum rules focus on the most critical problems in your code, including potential security holes, application crashes, and other important logic and design errors. Include this rule set in any custom rule set you create for your projects.

RULE	DESCRIPTION
CA1001	Types that own disposable fields should be disposable
CA1821	Remove empty finalizers
CA2213	Disposable fields should be disposed
CA2231	Overload operator equals on overriding `ValueType.Equals`

Managed Recommended Rules rule set for managed code

1/1/2020 • 2 minutes to read • Edit Online

Use the Microsoft Managed Recommended Rules rule set to focus on the most critical problems in your managed code, including potential security holes, application crashes, and other important logic and design errors. This rule set includes all the rules in the Managed Minimum Rules rule set.

Include this rule set in any custom rule set that you create for your projects.

RULE	DESCRIPTION
CA1001	Types that own disposable fields should be disposable
CA1009	Declare event handlers correctly
CA1016	Mark assemblies with AssemblyVersionAttribute
CA1033	Interface methods should be callable by child types
CA1049	Types that own native resources should be disposable
CA1060	Move P/Invokes to NativeMethods class
CA1061	Do not hide base class methods
CA1063	Implement IDisposable correctly
CA1065	Do not raise exceptions in unexpected locations
CA1301	Avoid duplicate accelerators
CA1400	P/Invoke entry points should exist
CA1401	P/Invokes should not be visible
CA1403	Auto layout types should not be COM visible
CA1404	Call GetLastError immediately after P/Invoke
CA1405	COM visible type base types should be COM visible
CA1410	COM registration methods should be matched
CA1415	Declare P/Invokes correctly
CA1821	Remove empty finalizers

RULE	DESCRIPTION
CA1900	Value type fields should be portable
CA1901	P/Invoke declarations should be portable
CA2002	Do not lock on objects with weak identity
CA2100	Review SQL queries for security vulnerabilities
CA2101	Specify marshaling for P/Invoke string arguments
CA2108	Review declarative security on value types
CA2111	Pointers should not be visible
CA2112	Secured types should not expose fields
CA2114	Method security should be a superset of type
CA2116	APTCA methods should only call APTCA methods
CA2117	APTCA types should only extend APTCA base types
CA2122	Do not indirectly expose methods with link demands
CA2123	Override link demands should be identical to base
CA2124	Wrap vulnerable finally clauses in outer try
CA2126	Type link demands require inheritance demands
CA2131	Security critical types may not participate in type equivalence
CA2132	Default constructors must be at least as critical as base type default constructors
CA2133	Delegates must bind to methods with consistent transparency
CA2134	Methods must keep consistent transparency when overriding base methods
CA2137	Transparent methods must contain only verifiable IL
CA2138	Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute
CA2140	Transparent code must not reference security critical items
CA2141	Transparent methods must not satisfy LinkDemands

RULE	DESCRIPTION
CA2146	Types must be at least as critical as their base types and interfaces
CA2147	Transparent methods may not use security asserts
CA2149	Transparent methods must not call into native code
CA2200	Rethrow to preserve stack details
CA2202	Do not dispose objects multiple times
CA2207	Initialize value type static fields inline
CA2212	Do not mark serviced components with WebMethod
CA2213	Disposable fields should be disposed
CA2214	Do not call overridable methods in constructors
CA2216	Disposable types should declare finalizer
CA2220	Finalizers should call base class finalizer
CA2229	Implement serialization constructors
CA2231	Overload operator equals on overriding ValueType.Equals
CA2232	Mark Windows Forms entry points with STAThread
CA2235	Mark all non-serializable fields
CA2236	Call base class methods on ISerializable types
CA2237	Mark ISerializable types with SerializableAttribute
CA2238	Implement serialization methods correctly
CA2240	Implement ISerializable correctly
CA2241	Provide correct arguments to formatting methods
CA2242	Test for NaN correctly

Mixed Minimum Rules rule set

4/20/2020 • 5 minutes to read • Edit Online

The Microsoft Mixed Minimum Rules focus on the most critical problems in C++ projects that support the Common Language Runtime, including potential security holes and application crashes.

Include this rule set in any custom rule set you create for your C++ projects that support the Common Language Runtime.

RULE	DESCRIPTION
C6001	Using Uninitialized Memory
C6011	Dereferencing Null Pointer
C6029	Use Of Unchecked Value
C6053	Zero Termination From Call
C6059	Bad Concatenation
C6063	Missing String Argument To Format Function
C6064	Missing Integer Argument To Format Function
C6066	Missing Pointer Argument To Format Function
C6067	Missing String Pointer Argument To Format Function
C6101	Returning uninitialized memory
C6200	Index Exceeds Buffer Maximum
C6201	Index Exceeds Stack Buffer Maximum
C6270	Missing Float Argument To Format Function
C6271	Extra Argument To Format Function
C6272	Non-Float Argument To Format Function
C6273	Non-Integer Argument To Format Function
C6274	Non-Character Argument To Format Function
C6276	Invalid String Cast
C6277	Invalid CreateProcess Call

RULE	DESCRIPTION
C6284	Invalid Object Argument To Format Function
C6290	Logical-Not Bitwise-And Precedence
C6291	Logical-Not Bitwise-Or Precedence
C6302	Invalid Character String Argument To Format Function
C6303	Invalid Wide Character String Argument To Format Function
C6305	Mismatched Size And Count Use
C6306	Incorrect Variable Argument Function Call
C6328	Potential Argument Type Mismatch
C6385	Read Overrun
C6386	Write Overrun
C6387	Invalid Parameter Value
C6500	Invalid Attribute Property
C6501	Conflicting Attribute Property Values
C6503	References Cannot Be Null
C6504	Null On Non-Pointer
C6505	MustCheck On Void
C6506	Buffer Size On Non-Pointer Or Array
C6508	Write Access On Constant
C6509	Return Used On Precondition
C6510	Null Terminated On Non-Pointer
C6511	MustCheck Must Be Yes Or No
C6513	Element Size Without Buffer Size
C6514	Buffer Size Exceeds Array Size
C6515	Buffer Size On Non-Pointer
C6516	No Properties On Attribute

RULE	DESCRIPTION
C6517	Valid Size On Non-Readable Buffer
C6518	Writable Size On Non-Writable Buffer
C6522	Invalid Size String Type
C6525	Invalid Size String Unreachable Location
C6527	Invalid annotation: 'NeedsRelease' property may not be used on values of void type
C6530	Unrecognized Format String Style
C6540	The use of attribute annotations on this function will invalidate all of its existing __declspec annotations
C6551	Invalid size specification: expression not parsable
C6552	Invalid Deref= or Notref=: expression not parsable
C6701	The value is not a valid Yes/No/Maybe value
C6702	The value is not a string value
C6703	The value is not a number
C6704	Unexpected Annotation Expression Error
C6705	Expected number of arguments for annotation does not match actual number of arguments for annotation
C6706	Unexpected Annotation Error for annotation
C28021	The parameter being annotated must be a pointer
C28182	Dereferencing NULL pointer. The pointer contains the same NULL value as another pointer did.
C28202	Illegal reference to non-static member
C28203	Ambiguous reference to class member.
C28205	_Success_ or _On_failure_ used in an illegal context
C28206	Left operand points to a struct, use '->'
C28207	Left operand is a struct, use '.'
C28210	Annotations for the __on_failure context must not be in explicit pre context

RULE	DESCRIPTION
C28211	Static context name expected for SAL_context
C28212	Pointer expression expected for annotation
C28213	The _Use_decl_annotations_ annotation must be used to reference, without modification, a prior declaration.
C28214	Attribute parameter names must be p1...p9
C28215	The typefix cannot be applied to a parameter that already has a typefix
C28216	The checkReturn annotation only applies to postconditions for the specific function parameter.
C28217	For function, the number of parameters to annotation does not match that found at file
C28218	For function parameter, the annotation's parameter does not match that found at file
C28219	Member of enumeration expected for annotation the parameter in the annotation
C28220	Integer expression expected for annotation the parameter in the annotation
C28221	String expression expected for the parameter in the annotation
C28222	__yes, __no, or __maybe expected for annotation
C28223	Did not find expected Token/identifier for annotation, parameter
C28224	Annotation requires parameters
C28225	Did not find the correct number of required parameters in annotation
C28226	Annotation cannot also be a PrimOp (in current declaration)
C28227	Annotation cannot also be a PrimOp (see prior declaration)
C28228	Annotation parameter: cannot use type in annotations
C28229	Annotation does not support parameters
C28230	The type of parameter has no member.
C28231	Annotation is only valid on array

RULE	DESCRIPTION
C28232	pre, post, or deref not applied to any annotation
C28233	pre, post, or deref applied to a block
C28234	__at expression does not apply to current function
C28235	The function cannot stand alone as an annotation
C28236	The annotation cannot be used in an expression
C28237	The annotation on parameter is no longer supported
C28238	The annotation on parameter has more than one of value, stringValue, and longValue. Use paramn=xxx
C28239	The annotation on parameter has both value, stringValue, or longValue; and paramn=xxx. Use only paramn=xxx
C28240	The annotation on parameter has param2 but no param1
C28241	The annotation for function on parameter is not recognized
C28243	The annotation for function on parameter requires more dereferences than the actual type annotated allows
C28245	The annotation for function annotates 'this' on a non-member-function
C28246	The parameter annotation for function does not match the type of the parameter
C28250	Inconsistent annotation for function: the prior instance has an error.
C28251	Inconsistent annotation for function: this instance has an error.
C28252	Inconsistent annotation for function: parameter has another annotations on this instance.
C28253	Inconsistent annotation for function: parameter has another annotations on this instance.
C28254	dynamic_cast<>() is not supported in annotations
C28262	A syntax error in the annotation was found in function, for annotation
C28263	A syntax error in a conditional annotation was found for Intrinsic annotation
C28267	A syntax error in the annotations was found annotation in the function.

RULE	DESCRIPTION
C28272	The annotation for function, parameter when examining is inconsistent with the function declaration
C28273	For function, the clues are inconsistent with the function declaration
C28275	The parameter to _Macro_value_ is null
C28279	For symbol, a 'begin' was found without a matching 'end'
C28280	For symbol, an 'end' was found without a matching 'begin'
C28282	Format Strings must be in preconditions
C28285	For function, syntax error in parameter
C28286	For function, syntax error near the end
C28287	For function, syntax Error in _At_() annotation (unrecognized parameter name)
C28288	For function, syntax Error in _At_() annotation (invalid parameter name)
C28289	For function: ReadableTo or WritableTo did not have a limit-spec as a parameter
C28290	the annotation for function contains more Externals than the actual number of parameters
C28291	post null/notnull at deref level 0 is meaningless for function.
C28300	Expression operands of incompatible types for operator
C28301	No annotations for first declaration of function.
C28302	An extra _Deref_ operator was found on annotation.
C28303	An ambiguous _Deref_ operator was found on annotation.
C28304	An improperly placed _Notref_ operator was found applied to token.
C28305	An error while parsing a token was discovered.
C28350	The annotation describes a situation that is not conditionally applicable.
C28351	The annotation describes where a dynamic value (a variable) cannot be used in the condition.
CA1001	Types that own disposable fields should be disposable

RULE	DESCRIPTION
CA1821	Remove empty finalizers
CA2213	Disposable fields should be disposed
CA2231	Overload operator equals on overriding ValueType.Equals

Mixed Recommended Rules rule set

4/20/2020 • 11 minutes to read • Edit Online

The Microsoft Mixed Recommended Rules focus on the most common and critical problems in your C++ projects that support the Common Language Runtime, including potential security holes, application crashes, and other important logic and design errors. This rule set includes all of the rules in the Mixed Minimum Rules rule set.

Include this rule set in any custom rule set you create for your C++ projects that support the Common Language Runtime.

RULE	DESCRIPTION
C6001	Using Uninitialized Memory
C6011	Dereferencing Null Pointer
C6029	Use Of Unchecked Value
C6031	Return Value Ignored
C6053	Zero Termination From Call
C6054	Zero Termination Missing
C6059	Bad Concatenation
C6063	Missing String Argument To Format Function
C6064	Missing Integer Argument To Format Function
C6066	Missing Pointer Argument To Format Function
C6067	Missing String Pointer Argument To Format Function
C6101	Returning uninitialized memory
C6200	Index Exceeds Buffer Maximum
C6201	Index Exceeds Stack Buffer Maximum
C6214	Invalid Cast HRESULT To BOOL
C6215	Invalid Cast BOOL To HRESULT
C6216	Invalid Compiler-Inserted Cast BOOL To HRESULT
C6217	Invalid HRESULT Test With NOT
C6220	Invalid HRESULT Compare To -1

RULE	DESCRIPTION
C6226	Invalid HRESULT Assignment To -1
C6230	Invalid HRESULT Use As Boolean
C6235	Non-Zero Constant With Logical-Or
C6236	Logical-Or With Non-Zero Constant
C6237	Zero With Logical-And Loses Side Effects
C6242	Local Unwind Forced
C6248	Creating Null DACL
C6250	Unreleased Address Descriptors
C6255	Unprotected Use Of Alloca
C6258	Using Terminate Thread
C6259	Dead Code In Bitwise-Or Limited Switch
C6260	Use Of Byte Arithmetic
C6262	Excessive Stack Usage
C6263	Using Alloca In Loop
C6268	Missing Parentheses In Cast
C6269	Pointer Dereference Ignored
C6270	Missing Float Argument To Format Function
C6271	Extra Argument To Format Function
C6272	Non-Float Argument To Format Function
C6273	Non-Integer Argument To Format Function
C6274	Non-Character Argument To Format Function
C6276	Invalid String Cast
C6277	Invalid CreateProcess Call
C6278	Array-New Scalar-Delete Mismatch
C6279	Scalar-New Array-Delete Mismatch

RULE	DESCRIPTION
C6280	Memory Allocation-Deallocation Mismatch
C6281	Bitwise Relation Precedence
C6282	Assignment Replaces Test
C6283	Primitive Array-New Scalar-Delete Mismatch
C6284	Invalid Object Argument To Format Function
C6285	Logical-Or Of Constants
C6286	Non-Zero Logical-Or Losing Side Effects
C6287	Redundant Test
C6288	Mutual Inclusion Over Logical-And Is False
C6289	Mutual Exclusion Over Logical-Or Is True
C6290	Logical-Not Bitwise-And Precedence
C6291	Logical-Not Bitwise-Or Precedence
C6292	Loop Counts Up From Maximum
C6293	Loop Counts Down From Minimum
C6294	Loop Body Never Executed
C6295	Infinite Loop
C6296	Loop Only Executed Once
C6297	Result Of Shift Cast To Larger Size
C6299	Bitfield To Boolean Comparison
C6302	Invalid Character String Argument To Format Function
C6303	Invalid Wide Character String Argument To Format Function
C6305	Mismatched Size And Count Use
C6306	Incorrect Variable Argument Function Call
C6308	Realloc Leak
C6310	Illegal Exception Filter Constant

RULE	DESCRIPTION
C6312	Exception Continue Execution Loop
C6314	Bitwise-Or Precedence
C6317	Not Not Complement
C6318	Exception Continue Search
C6319	Ignored By Comma
C6324	String Copy Instead Of String Compare
C6328	Potential Argument Type Mismatch
C6331	VirtualFree Invalid Flags
C6332	VirtualFree Invalid Parameter
C6333	VirtualFree Invalid Size
C6335	Leaking Process Handle
C6381	Shutdown Information Missing
C6383	Element-Count Byte-Count Buffer Overrun
C6384	Pointer Size Division
C6385	Read Overrun
C6386	Write Overrun
C6387	Invalid Parameter Value
C6388	Invalid Parameter Value
C6500	Invalid Attribute Property
C6501	Conflicting Attribute Property Values
C6503	References Cannot Be Null
C6504	Null On Non-Pointer
C6505	MustCheck On Void
C6506	Buffer Size On Non-Pointer Or Array
C6508	Write Access On Constant

RULE	DESCRIPTION
C6509	Return Used On Precondition
C6510	Null Terminated On Non-Pointer
C6511	MustCheck Must Be Yes Or No
C6513	Element Size Without Buffer Size
C6514	Buffer Size Exceeds Array Size
C6515	Buffer Size On Non-Pointer
C6516	No Properties On Attribute
C6517	Valid Size On Non-Readable Buffer
C6518	Writable Size On Non-Writable Buffer
C6522	Invalid Size String Type
C6525	Invalid Size String Unreachable Location
C6527	Invalid annotation: 'NeedsRelease' property may not be used on values of void type
C6530	Unrecognized Format String Style
C6540	The use of attribute annotations on this function will invalidate all of its existing __declspec annotations
C6551	Invalid size specification: expression not parsable
C6552	Invalid Deref= or Notref=: expression not parsable
C6701	The value is not a valid Yes/No/Maybe value
C6702	The value is not a string value
C6703	The value is not a number
C6704	Unexpected Annotation Expression Error
C6705	Expected number of arguments for annotation does not match actual number of arguments for annotation
C6706	Unexpected Annotation Error for annotation
C6995	Failed to save XML Log file
C26100	Race condition

RULE	DESCRIPTION
C26101	Failing to use interlocked operation properly
C26110	Caller failing to hold lock
C26111	Caller failing to release lock
C26112	Caller cannot hold any lock
C26115	Failing to release lock
C26116	Failing to acquire or to hold lock
C26117	Releasing unheld lock
C26140	Concurrency SAL annotation error
C28020	The expression is not true at this call
C28021	The parameter being annotated must be a pointer
C28022	The function class(es) on this function do not match the function class(es) on the typedef used to define it.
C28023	The function being assigned or passed should have a _Function_class_ annotation for at least one of the class(es)
C28024	The function pointer being assigned to is annotated with the function class, which is not contained in the function class(es) list.
C28039	The type of actual parameter should exactly match the type
C28112	A variable which is accessed via an Interlocked function must always be accessed via an Interlocked function.
C28113	Accessing a local variable via an Interlocked function
C28125	The function must be called from within a try/except block
C28137	The variable argument should instead be a (literal) constant
C28138	The constant argument should instead be variable
C28159	Consider using another function instead.
C28160	Error annotation
C28163	The function should never be called from within a try/except block

RULE	DESCRIPTION
C28164	The argument is being passed to a function that expects a pointer to an object (not a pointer to a pointer)
C28182	Dereferencing NULL pointer. The pointer contains the same NULL value as another pointer did.
C28183	The argument could be one value, and is a copy of the value found in the pointer
C28193	The variable holds a value that must be examined
C28196	The requirement is not satisfied. (The expression does not evaluate to true.)
C28202	Illegal reference to non-static member
C28203	Ambiguous reference to class member.
C28205	_Success_ or _On_failure_ used in an illegal context
C28206	Left operand points to a struct, use '->'
C28207	Left operand is a struct, use '.'
C28209	The declaration for symbol has a conflicting declaration
C28210	Annotations for the __on_failure context must not be in explicit pre context
C28211	Static context name expected for SAL_context
C28212	Pointer expression expected for annotation
C28213	The _Use_decl_annotations_ annotation must be used to reference, without modification, a prior declaration.
C28214	Attribute parameter names must be p1...p9
C28215	The typefix cannot be applied to a parameter that already has a typefix
C28216	The checkReturn annotation only applies to postconditions for the specific function parameter.
C28217	For function, the number of parameters to annotation does not match that found at file
C28218	For function parameter, the annotation's parameter does not match that found at file
C28219	Member of enumeration expected for annotation the parameter in the annotation

RULE	DESCRIPTION
C28220	Integer expression expected for annotation the parameter in the annotation
C28221	String expression expected for the parameter in the annotation
C28222	__yes, __no, or __maybe expected for annotation
C28223	Did not find expected Token/identifier for annotation, parameter
C28224	Annotation requires parameters
C28225	Did not find the correct number of required parameters in annotation
C28226	Annotation cannot also be a PrimOp (in current declaration)
C28227	Annotation cannot also be a PrimOp (see prior declaration)
C28228	Annotation parameter: cannot use type in annotations
C28229	Annotation does not support parameters
C28230	The type of parameter has no member.
C28231	Annotation is only valid on array
C28232	pre, post, or deref not applied to any annotation
C28233	pre, post, or deref applied to a block
C28234	__at expression does not apply to current function
C28235	The function cannot stand alone as an annotation
C28236	The annotation cannot be used in an expression
C28237	The annotation on parameter is no longer supported
C28238	The annotation on parameter has more than one of value, stringValue, and longValue. Use paramn=xxx
C28239	The annotation on parameter has both value, stringValue, or longValue; and paramn=xxx. Use only paramn=xxx
C28240	The annotation on parameter has param2 but no param1
C28241	The annotation for function on parameter is not recognized

RULE	DESCRIPTION
C28243	The annotation for function on parameter requires more dereferences than the actual type annotated allows
C28244	The annotation for function has an unparsable parameter/external annotation
C28245	The annotation for function annotates 'this' on a non-member-function
C28246	The parameter annotation for function does not match the type of the parameter
C28250	Inconsistent annotation for function: the prior instance has an error.
C28251	Inconsistent annotation for function: this instance has an error.
C28252	Inconsistent annotation for function: parameter has another annotations on this instance.
C28253	Inconsistent annotation for function: parameter has another annotations on this instance.
C28254	dynamic_cast<>() is not supported in annotations
C28262	A syntax error in the annotation was found in function, for annotation
C28263	A syntax error in a conditional annotation was found for Intrinsic annotation
C28267	A syntax error in the annotations was found annotation in the function.
C28272	The annotation for function, parameter when examining is inconsistent with the function declaration
C28273	For function, the clues are inconsistent with the function declaration
C28275	The parameter to _Macro_value_ is null
C28279	For symbol, a 'begin' was found without a matching 'end'
C28280	For symbol, an 'end' was found without a matching 'begin'
C28282	Format Strings must be in preconditions
C28285	For function, syntax error in parameter
C28286	For function, syntax error near the end

RULE	DESCRIPTION
C28287	For function, syntax Error in _At_() annotation (unrecognized parameter name)
C28288	For function, syntax Error in _At_() annotation (invalid parameter name)
C28289	For function: ReadableTo or WritableTo did not have a limit-spec as a parameter
C28290	the annotation for function contains more Externals than the actual number of parameters
C28291	post null/notnull at deref level 0 is meaningless for function.
C28300	Expression operands of incompatible types for operator
C28301	No annotations for first declaration of function.
C28302	An extra _Deref_ operator was found on annotation.
C28303	An ambiguous _Deref_ operator was found on annotation.
C28304	An improperly placed _Notref_ operator was found applied to token.
C28305	An error while parsing a token was discovered.
C28306	The annotation on parameter is obsolescent
C28307	The annotation on parameter is obsolescent
C28350	The annotation describes a situation that is not conditionally applicable.
C28351	The annotation describes where a dynamic value (a variable) cannot be used in the condition.
CA1001	Types that own disposable fields should be disposable
CA1009	Declare event handlers correctly
CA1016	Mark assemblies with AssemblyVersionAttribute
CA1033	Interface methods should be callable by child types
CA1049	Types that own native resources should be disposable
CA1060	Move P/Invokes to NativeMethods class
CA1061	Do not hide base class methods

RULE	DESCRIPTION
CA1063	Implement IDisposable correctly
CA1065	Do not raise exceptions in unexpected locations
CA1301	Avoid duplicate accelerators
CA1400	P/Invoke entry points should exist
CA1401	P/Invokes should not be visible
CA1403	Auto layout types should not be COM visible
CA1404	Call GetLastError immediately after P/Invoke
CA1405	COM visible type base types should be COM visible
CA1410	COM registration methods should be matched
CA1415	Declare P/Invokes correctly
CA1821	Remove empty finalizers
CA1900	Value type fields should be portable
CA1901	P/Invoke declarations should be portable
CA2002	Do not lock on objects with weak identity
CA2100	Review SQL queries for security vulnerabilities
CA2101	Specify marshaling for P/Invoke string arguments
CA2108	Review declarative security on value types
CA2111	Pointers should not be visible
CA2112	Secured types should not expose fields
CA2114	Method security should be a superset of type
CA2116	APTCA methods should only call APTCA methods
CA2117	APTCA types should only extend APTCA base types
CA2122	Do not indirectly expose methods with link demands
CA2123	Override link demands should be identical to base
CA2124	Wrap vulnerable finally clauses in outer try

RULE	DESCRIPTION
CA2126	Type link demands require inheritance demands
CA2131	Security critical types may not participate in type equivalence
CA2132	Default constructors must be at least as critical as base type default constructors
CA2133	Delegates must bind to methods with consistent transparency
CA2134	Methods must keep consistent transparency when overriding base methods
CA2137	Transparent methods must contain only verifiable IL
CA2138	Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute
CA2140	Transparent code must not reference security critical items
CA2141	Transparent methods must not satisfy LinkDemands
CA2146	Types must be at least as critical as their base types and interfaces
CA2147	Transparent methods may not use security asserts
CA2149	Transparent methods must not call into native code
CA2200	Rethrow to preserve stack details
CA2202	Do not dispose objects multiple times
CA2207	Initialize value type static fields inline
CA2212	Do not mark serviced components with WebMethod
CA2213	Disposable fields should be disposed
CA2214	Do not call overridable methods in constructors
CA2216	Disposable types should declare finalizer
CA2220	Finalizers should call base class finalizer
CA2229	Implement serialization constructors
CA2231	Overload operator equals on overriding ValueType.Equals
CA2232	Mark Windows Forms entry points with STAThread
CA2235	Mark all non-serializable fields

RULE	DESCRIPTION
CA2236	Call base class methods on ISerializable types
CA2237	Mark ISerializable types with SerializableAttribute
CA2238	Implement serialization methods correctly
CA2240	Implement ISerializable correctly
CA2241	Provide correct arguments to formatting methods
CA2242	Test for NaN correctly

Native Minimum Rules rule set

4/20/2020 • 5 minutes to read • Edit Online

The Microsoft Native Minimum Rules focus on the most critical problems in native code, including potential security holes and application crashes.

Include this rule set in any custom rule set you create for native projects.

RULE	DESCRIPTION
C6001	Using Uninitialized Memory
C6011	Dereferencing Null Pointer
C6029	Use Of Unchecked Value
C6053	Zero Termination From Call
C6059	Bad Concatenation
C6063	Missing String Argument To Format Function
C6064	Missing Integer Argument To Format Function
C6066	Missing Pointer Argument To Format Function
C6067	Missing String Pointer Argument To Format Function
C6101	Returning uninitialized memory
C6200	Index Exceeds Buffer Maximum
C6201	Index Exceeds Stack Buffer Maximum
C6270	Missing Float Argument To Format Function
C6271	Extra Argument To Format Function
C6272	Non-Float Argument To Format Function
C6273	Non-Integer Argument To Format Function
C6274	Non-Character Argument To Format Function
C6276	Invalid String Cast
C6277	Invalid CreateProcess Call
C6284	Invalid Object Argument To Format Function

RULE	DESCRIPTION
C6290	Logical-Not Bitwise-And Precedence
C6291	Logical-Not Bitwise-Or Precedence
C6302	Invalid Character String Argument To Format Function
C6303	Invalid Wide Character String Argument To Format Function
C6305	Mismatched Size And Count Use
C6306	Incorrect Variable Argument Function Call
C6328	Potential Argument Type Mismatch
C6385	Read Overrun
C6386	Write Overrun
C6387	Invalid Parameter Value
C6500	Invalid Attribute Property
C6501	Conflicting Attribute Property Values
C6503	References Cannot Be Null
C6504	Null On Non-Pointer
C6505	MustCheck On Void
C6506	Buffer Size On Non-Pointer Or Array
C6508	Write Access On Constant
C6509	Return Used On Precondition
C6510	Null Terminated On Non-Pointer
C6511	MustCheck Must Be Yes Or No
C6513	Element Size Without Buffer Size
C6514	Buffer Size Exceeds Array Size
C6515	Buffer Size On Non-Pointer
C6516	No Properties On Attribute
C6517	Valid Size On Non-Readable Buffer

RULE	DESCRIPTION
C6518	Writable Size On Non-Writable Buffer
C6522	Invalid Size String Type
C6525	Invalid Size String Unreachable Location
C6527	Invalid annotation: 'NeedsRelease' property may not be used on values of void type
C6530	Unrecognized Format String Style
C6540	The use of attribute annotations on this function will invalidate all of its existing __declspec annotations
C6551	Invalid size specification: expression not parsable
C6552	Invalid Deref= or Notref=: expression not parsable
C6701	The value is not a valid Yes/No/Maybe value
C6702	The value is not a string value
C6703	The value is not a number
C6704	Unexpected Annotation Expression Error
C6705	Expected number of arguments for annotation does not match actual number of arguments for annotation
C6706	Unexpected Annotation Error for annotation
C26450	RESULT_OF_ARITHMETIC_OPERATION_PROVABLY_LOSSY
C26451	RESULT_OF_ARITHMETIC_OPERATION_CAST_TO_LARGER_SIZE
C26452	SHIFT_COUNT_NEGATIVE_OR_TOO_BIG
C26453	LEFTSHIFT_NEGATIVE_SIGNED_NUMBER
C26454	RESULT_OF_ARITHMETIC_OPERATION_NEGATIVE_UNSIGNED
C26495	MEMBER_UNINIT
C28021	The parameter being annotated must be a pointer
C28182	Dereferencing NULL pointer. The pointer contains the same NULL value as another pointer did.
C28202	Illegal reference to non-static member
C28203	Ambiguous reference to class member.

RULE	DESCRIPTION
C28205	_Success_ or _On_failure_ used in an illegal context
C28206	Left operand points to a struct, use '->'
C28207	Left operand is a struct, use '.'
C28210	Annotations for the __on_failure context must not be in explicit pre context
C28211	Static context name expected for SAL_context
C28212	Pointer expression expected for annotation
C28213	The _Use_decl_annotations_ annotation must be used to reference, without modification, a prior declaration.
C28214	Attribute parameter names must be p1...p9
C28215	The typefix cannot be applied to a parameter that already has a typefix
C28216	The checkReturn annotation only applies to postconditions for the specific function parameter.
C28217	For function, the number of parameters to annotation does not match that found at file
C28218	For function parameter, the annotation's parameter does not match that found at file
C28219	Member of enumeration expected for annotation the parameter in the annotation
C28220	Integer expression expected for annotation the parameter in the annotation
C28221	String expression expected for the parameter in the annotation
C28222	__yes, __no, or __maybe expected for annotation
C28223	Did not find expected Token/identifier for annotation, parameter
C28224	Annotation requires parameters
C28225	Did not find the correct number of required parameters in annotation
C28226	Annotation cannot also be a PrimOp (in current declaration)
C28227	Annotation cannot also be a PrimOp (see prior declaration)

RULE	DESCRIPTION
C28228	Annotation parameter: cannot use type in annotations
C28229	Annotation does not support parameters
C28230	The type of parameter has no member.
C28231	Annotation is only valid on array
C28232	pre, post, or deref not applied to any annotation
C28233	pre, post, or deref applied to a block
C28234	__at expression does not apply to current function
C28235	The function cannot stand alone as an annotation
C28236	The annotation cannot be used in an expression
C28237	The annotation on parameter is no longer supported
C28238	The annotation on parameter has more than one of value, stringValue, and longValue. Use paramn=xxx
C28239	The annotation on parameter has both value, stringValue, or longValue; and paramn=xxx. Use only paramn=xxx
C28240	The annotation on parameter has param2 but no param1
C28241	The annotation for function on parameter is not recognized
C28243	The annotation for function on parameter requires more dereferences than the actual type annotated allows
C28245	The annotation for function annotates 'this' on a non-member-function
C28246	The parameter annotation for function does not match the type of the parameter
C28250	Inconsistent annotation for function: the prior instance has an error.
C28251	Inconsistent annotation for function: this instance has an error.
C28252	Inconsistent annotation for function: parameter has another annotations on this instance.
C28253	Inconsistent annotation for function: parameter has another annotations on this instance.
C28254	dynamic_cast<>() is not supported in annotations

RULE	DESCRIPTION
C28262	A syntax error in the annotation was found in function, for annotation
C28263	A syntax error in a conditional annotation was found for Intrinsic annotation
C28267	A syntax error in the annotations was found annotation in the function.
C28272	The annotation for function, parameter when examining is inconsistent with the function declaration
C28273	For function, the clues are inconsistent with the function declaration
C28275	The parameter to _Macro_value_ is null
C28279	For symbol, a 'begin' was found without a matching 'end'
C28280	For symbol, an 'end' was found without a matching 'begin'
C28282	Format Strings must be in preconditions
C28285	For function, syntax error in parameter
C28286	For function, syntax error near the end
C28287	For function, syntax Error in _At_() annotation (unrecognized parameter name)
C28288	For function, syntax Error in _At_() annotation (invalid parameter name)
C28289	For function: ReadableTo or WritableTo did not have a limit-spec as a parameter
C28290	the annotation for function contains more Externals than the actual number of parameters
C28291	post null/notnull at deref level 0 is meaningless for function.
C28300	Expression operands of incompatible types for operator
C28301	No annotations for first declaration of function.
C28302	An extra _Deref_ operator was found on annotation.
C28303	An ambiguous _Deref_ operator was found on annotation.
C28304	An improperly placed _Notref_ operator was found applied to token.

RULE	DESCRIPTION
C28305	An error while parsing a token was discovered.
C28350	The annotation describes a situation that is not conditionally applicable.
C28351	The annotation describes where a dynamic value (a variable) cannot be used in the condition.

Native Recommended Rules rule set

4/20/2020 • 8 minutes to read • Edit Online

The Native Recommended Rules focus on the most critical and common problems in native code, including potential security holes and application crashes. This rule set includes all of the rules in the Native Minimum Rules rule set.

Include this rule set in any custom rule set you create for native projects.

RULE	DESCRIPTION
C6001	Using Uninitialized Memory
C6011	Dereferencing Null Pointer
C6029	Use Of Unchecked Value
C6031	Return Value Ignored
C6053	Zero Termination From Call
C6054	Zero Termination Missing
C6059	Bad Concatenation
C6063	Missing String Argument To Format Function
C6064	Missing Integer Argument To Format Function
C6066	Missing Pointer Argument To Format Function
C6067	Missing String Pointer Argument To Format Function
C6101	Returning uninitialized memory
C6200	Index Exceeds Buffer Maximum
C6201	Index Exceeds Stack Buffer Maximum
C6214	Invalid Cast HRESULT To BOOL
C6215	Invalid Cast BOOL To HRESULT
C6216	Invalid Compiler-Inserted Cast BOOL To HRESULT
C6217	Invalid HRESULT Test With NOT
C6220	Invalid HRESULT Compare To -1

RULE	DESCRIPTION
C6226	Invalid HRESULT Assignment To -1
C6230	Invalid HRESULT Use As Boolean
C6235	Non-Zero Constant With Logical-Or
C6236	Logical-Or With Non-Zero Constant
C6237	Zero With Logical-And Loses Side Effects
C6242	Local Unwind Forced
C6248	Creating Null DACL
C6250	Unreleased Address Descriptors
C6255	Unprotected Use Of Alloca
C6258	Using Terminate Thread
C6259	Dead Code In Bitwise-Or Limited Switch
C6260	Use Of Byte Arithmetic
C6262	Excessive Stack Usage
C6263	Using Alloca In Loop
C6268	Missing Parentheses In Cast
C6269	Pointer Dereference Ignored
C6270	Missing Float Argument To Format Function
C6271	Extra Argument To Format Function
C6272	Non-Float Argument To Format Function
C6273	Non-Integer Argument To Format Function
C6274	Non-Character Argument To Format Function
C6276	Invalid String Cast
C6277	Invalid CreateProcess Call
C6278	Array-New Scalar-Delete Mismatch
C6279	Scalar-New Array-Delete Mismatch

RULE	DESCRIPTION
C6280	Memory Allocation-Deallocation Mismatch
C6281	Bitwise Relation Precedence
C6282	Assignment Replaces Test
C6283	Primitive Array-New Scalar-Delete Mismatch
C6284	Invalid Object Argument To Format Function
C6285	Logical-Or Of Constants
C6286	Non-Zero Logical-Or Losing Side Effects
C6287	Redundant Test
C6288	Mutual Inclusion Over Logical-And Is False
C6289	Mutual Exclusion Over Logical-Or Is True
C6290	Logical-Not Bitwise-And Precedence
C6291	Logical-Not Bitwise-Or Precedence
C6292	Loop Counts Up From Maximum
C6293	Loop Counts Down From Minimum
C6294	Loop Body Never Executed
C6295	Infinite Loop
C6296	Loop Only Executed Once
C6297	Result Of Shift Cast To Larger Size
C6299	Bitfield To Boolean Comparison
C6302	Invalid Character String Argument To Format Function
C6303	Invalid Wide Character String Argument To Format Function
C6305	Mismatched Size And Count Use
C6306	Incorrect Variable Argument Function Call
C6308	Realloc Leak
C6310	Illegal Exception Filter Constant

RULE	DESCRIPTION
C6312	Exception Continue Execution Loop
C6314	Bitwise-Or Precedence
C6317	Not Not Complement
C6318	Exception Continue Search
C6319	Ignored By Comma
C6324	String Copy Instead Of String Compare
C6328	Potential Argument Type Mismatch
C6331	VirtualFree Invalid Flags
C6332	VirtualFree Invalid Parameter
C6333	VirtualFree Invalid Size
C6335	Leaking Process Handle
C6381	Shutdown Information Missing
C6383	Element-Count Byte-Count Buffer Overrun
C6384	Pointer Size Division
C6385	Read Overrun
C6386	Write Overrun
C6387	Invalid Parameter Value
C6388	Invalid Parameter Value
C6500	Invalid Attribute Property
C6501	Conflicting Attribute Property Values
C6503	References Cannot Be Null
C6504	Null On Non-Pointer
C6505	MustCheck On Void
C6506	Buffer Size On Non-Pointer Or Array
C6508	Write Access On Constant

RULE	DESCRIPTION
C6509	Return Used On Precondition
C6510	Null Terminated On Non-Pointer
C6511	MustCheck Must Be Yes Or No
C6513	Element Size Without Buffer Size
C6514	Buffer Size Exceeds Array Size
C6515	Buffer Size On Non-Pointer
C6516	No Properties On Attribute
C6517	Valid Size On Non-Readable Buffer
C6518	Writable Size On Non-Writable Buffer
C6522	Invalid Size String Type
C6525	Invalid Size String Unreachable Location
C6527	Invalid annotation: 'NeedsRelease' property may not be used on values of void type
C6530	Unrecognized Format String Style
C6540	The use of attribute annotations on this function will invalidate all of its existing __declspec annotations
C6551	Invalid size specification: expression not parsable
C6552	Invalid Deref= or Notref=: expression not parsable
C6701	The value is not a valid Yes/No/Maybe value
C6702	The value is not a string value
C6703	The value is not a number
C6704	Unexpected Annotation Expression Error
C6705	Expected number of arguments for annotation does not match actual number of arguments for annotation
C6706	Unexpected Annotation Error for annotation
C6995	Failed to save XML Log file
C26100	Race condition

RULE	DESCRIPTION
C26101	Failing to use interlocked operation properly
C26110	Caller failing to hold lock
C26111	Caller failing to release lock
C26112	Caller cannot hold any lock
C26115	Failing to release lock
C26116	Failing to acquire or to hold lock
C26117	Releasing unheld lock
C26140	Concurrency SAL annotation error
C26441	NO_UNNAMED_GUARDS
C26444	NO_UNNAMED_RAII_OBJECTS
C26498	USE_CONSTEXPR_FOR_FUNCTIONCALL
C28020	The expression is not true at this call
C28021	The parameter being annotated must be a pointer
C28022	The function class(es) on this function do not match the function class(es) on the typedef used to define it.
C28023	The function being assigned or passed should have a _Function_class_ annotation for at least one of the class(es)
C28024	The function pointer being assigned to is annotated with the function class, which is not contained in the function class(es) list.
C28039	The type of actual parameter should exactly match the type
C28112	A variable which is accessed via an Interlocked function must always be accessed via an Interlocked function.
C28113	Accessing a local variable via an Interlocked function
C28125	The function must be called from within a try/except block
C28137	The variable argument should instead be a (literal) constant
C28138	The constant argument should instead be variable
C28159	Consider using another function instead.

RULE	DESCRIPTION
C28160	Error annotation
C28163	The function should never be called from within a try/except block
C28164	The argument is being passed to a function that expects a pointer to an object (not a pointer to a pointer)
C28182	Dereferencing NULL pointer. The pointer contains the same NULL value as another pointer did.
C28183	The argument could be one value, and is a copy of the value found in the pointer
C28193	The variable holds a value that must be examined
C28196	The requirement is not satisfied. (The expression does not evaluate to true.)
C28202	Illegal reference to non-static member
C28203	Ambiguous reference to class member.
C28205	_Success_ or _On_failure_ used in an illegal context
C28206	Left operand points to a struct, use '->'
C28207	Left operand is a struct, use '.'
C28209	The declaration for symbol has a conflicting declaration
C28210	Annotations for the __on_failure context must not be in explicit pre context
C28211	Static context name expected for SAL_context
C28212	Pointer expression expected for annotation
C28213	The _Use_decl_annotations_ annotation must be used to reference, without modification, a prior declaration.
C28214	Attribute parameter names must be p1...p9
C28215	The typefix cannot be applied to a parameter that already has a typefix
C28216	The checkReturn annotation only applies to postconditions for the specific function parameter.
C28217	For function, the number of parameters to annotation does not match that found at file

RULE	DESCRIPTION
C28218	For function parameter, the annotation's parameter does not match that found at file
C28219	Member of enumeration expected for annotation the parameter in the annotation
C28220	Integer expression expected for annotation the parameter in the annotation
C28221	String expression expected for the parameter in the annotation
C28222	__yes, __no, or __maybe expected for annotation
C28223	Did not find expected Token/identifier for annotation, parameter
C28224	Annotation requires parameters
C28225	Did not find the correct number of required parameters in annotation
C28226	Annotation cannot also be a PrimOp (in current declaration)
C28227	Annotation cannot also be a PrimOp (see prior declaration)
C28228	Annotation parameter: cannot use type in annotations
C28229	Annotation does not support parameters
C28230	The type of parameter has no member.
C28231	Annotation is only valid on array
C28232	pre, post, or deref not applied to any annotation
C28233	pre, post, or deref applied to a block
C28234	__at expression does not apply to current function
C28235	The function cannot stand alone as an annotation
C28236	The annotation cannot be used in an expression
C28237	The annotation on parameter is no longer supported
C28238	The annotation on parameter has more than one of value, stringValue, and longValue. Use paramn=xxx
C28239	The annotation on parameter has both value, stringValue, or longValue; and paramn=xxx. Use only paramn=xxx

RULE	DESCRIPTION
C28240	The annotation on parameter has param2 but no param1
C28241	The annotation for function on parameter is not recognized
C28243	The annotation for function on parameter requires more dereferences than the actual type annotated allows
C28244	The annotation for function has an unparsable parameter/external annotation
C28245	The annotation for function annotates 'this' on a non-member-function
C28246	The parameter annotation for function does not match the type of the parameter
C28250	Inconsistent annotation for function: the prior instance has an error.
C28251	Inconsistent annotation for function: this instance has an error.
C28252	Inconsistent annotation for function: parameter has another annotations on this instance.
C28253	Inconsistent annotation for function: parameter has another annotations on this instance.
C28254	dynamic_cast<>() is not supported in annotations
C28262	A syntax error in the annotation was found in function, for annotation
C28263	A syntax error in a conditional annotation was found for Intrinsic annotation
C28267	A syntax error in the annotations was found annotation in the function.
C28272	The annotation for function, parameter when examining is inconsistent with the function declaration
C28273	For function, the clues are inconsistent with the function declaration
C28275	The parameter to _Macro_value_ is null
C28279	For symbol, a 'begin' was found without a matching 'end'
C28280	For symbol, an 'end' was found without a matching 'begin'
C28282	Format Strings must be in preconditions

RULE	DESCRIPTION
C28285	For function, syntax error in parameter
C28286	For function, syntax error near the end
C28287	For function, syntax Error in _At_() annotation (unrecognized parameter name)
C28288	For function, syntax Error in _At_() annotation (invalid parameter name)
C28289	For function: ReadableTo or WritableTo did not have a limit-spec as a parameter
C28290	the annotation for function contains more Externals than the actual number of parameters
C28291	post null/notnull at deref level 0 is meaningless for function.
C28300	Expression operands of incompatible types for operator
C28301	No annotations for first declaration of function.
C28302	An extra _Deref_ operator was found on annotation.
C28303	An ambiguous _Deref_ operator was found on annotation.
C28304	An improperly placed _Notref_ operator was found applied to token.
C28305	An error while parsing a token was discovered.
C28306	The annotation on parameter is obsolescent
C28307	The annotation on parameter is obsolescent
C28350	The annotation describes a situation that is not conditionally applicable.
C28351	The annotation describes where a dynamic value (a variable) cannot be used in the condition.

Security Rules rule set for managed code

1/14/2020 • 2 minutes to read • Edit Online

Use the Microsoft Security Rules rule set for legacy code analysis to maximize the number of potential security issues that are reported.

RULE	DESCRIPTION
CA2100	Review SQL queries for security vulnerabilities
CA2102	Catch non-CLSCompliant exceptions in general handlers
CA2103	Review imperative security
CA2104	Do not declare read only mutable reference types
CA2105	Array fields should not be read only
CA2106	Secure asserts
CA2107	Review deny and permit only usage
CA2108	Review declarative security on value types
CA2109	Review visible event handlers
CA2111	Pointers should not be visible
CA2112	Secured types should not expose fields
CA2114	Method security should be a superset of type
CA2115	Call GC.KeepAlive when using native resources
CA2116	APTCA methods should only call APTCA methods
CA2117	APTCA types should only extend APTCA base types
CA2118	Review SuppressUnmanagedCodeSecurityAttribute usage
CA2119	Seal methods that satisfy private interfaces
CA2120	Secure serialization constructors
CA2121	Static constructors should be private
CA2122	Do not indirectly expose methods with link demands
CA2123	Override link demands should be identical to base

RULE	DESCRIPTION
CA2124	Wrap vulnerable finally clauses in outer try
CA2126	Type link demands require inheritance demands
CA2130	Security critical constants should be transparent
CA2131	Security critical types may not participate in type equivalence
CA2132	Default constructors must be at least as critical as base type default constructors
CA2133	Delegates must bind to methods with consistent transparency
CA2134	Methods must keep consistent transparency when overriding base methods
CA2135	Level 2 assemblies should not contain LinkDemands
CA2136	Members should not have conflicting transparency annotations
CA2137	Transparent methods must contain only verifiable IL
CA2138	Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute
CA2139	Transparent methods may not use the HandleProcessCorruptingExceptions attribute
CA2140	Transparent code must not reference security critical items
CA2141	Transparent methods must not satisfy LinkDemands
CA2142	Transparent code should not be protected with LinkDemands
CA2143	Transparent methods should not use security demands
CA2144	Transparent code should not load assemblies from byte arrays
CA2145	Transparent methods should not be decorated with the SuppressUnmanagedCodeSecurityAttribute
CA2146	Types must be at least as critical as their base types and interfaces
CA2147	Transparent methods may not use security asserts
CA2149	Transparent methods must not call into native code
CA2210	Assemblies should have valid strong names

RULE	DESCRIPTION
CA2300	Do not use insecure deserializer BinaryFormatter
CA2301	Do not call BinaryFormatter.Deserialize without first setting BinaryFormatter.Binder
CA2302	Ensure BinaryFormatter.Binder is set before calling BinaryFormatter.Deserialize
CA2305	Do not use insecure deserializer LosFormatter
CA2310	Do not use insecure deserializer NetDataContractSerializer
CA2311	Do not deserialize without first setting NetDataContractSerializer.Binder
CA2312	Ensure NetDataContractSerializer.Binder is set before deserializing
CA2315	Do not use insecure deserializer ObjectStateFormatter
CA2321	Do not deserialize with JavaScriptSerializer using a SimpleTypeResolver
CA2322	Ensure JavaScriptSerializer is not initialized with SimpleTypeResolver before deserializing
CA3001	Review code for SQL injection vulnerabilities
CA3002	Review code for XSS vulnerabilities
CA3003	Review code for file path injection vulnerabilities
CA3004	Review code for information disclosure vulnerabilities
CA3005	Review code for LDAP injection vulnerabilities
CA3006	Review code for process command injection vulnerabilities
CA3007	Review code for open redirect vulnerabilities
CA3008	Review code for XPath injection vulnerabilities
CA3009	Review code for XML injection vulnerabilities
CA3010	Review code for XAML injection vulnerabilities
CA3011	Review code for DLL injection vulnerabilities
CA3012	Review code for regex injection vulnerabilities
CA5358	Do Not Use Unsafe Cipher Modes

RULE	DESCRIPTION
CA5403	Do not hard-code certificate

Suppress code analysis warnings

3/5/2020 • 7 minutes to read • Edit Online

It is often useful to indicate that a warning is not applicable. This indicates to team members that the code was reviewed, and that the warning can be suppressed. In-source suppression (ISS) uses the SuppressMessageAttribute attribute to suppress a warning. The attribute can be placed close to the code segment that generated the warning. You can add the SuppressMessageAttribute attribute to the source file by typing it in, or you can use the shortcut menu on a warning in the **Error List** to add it automatically.

The SuppressMessageAttribute attribute is a conditional attribute, which is included in the IL metadata of your managed code assembly, only if the CODE_ANALYSIS compilation symbol is defined at compile time.

In C++/CLI, use the macros CA_SUPPRESS_MESSAGE or CA_GLOBAL_SUPPRESS_MESSAGE in the header file to add the attribute.

> **NOTE**
>
> You should not use in-source suppressions on release builds, to prevent shipping the in-source suppression metadata accidentally. Additionally, because of the processing cost of in-source suppression, the performance of your application can be degraded.

> **NOTE**
>
> If you migrate a project to Visual Studio 2017, you might suddenly be faced with a large number of code analysis warnings. If you aren't ready to fix the warnings, you can suppress all of them by choosing **Analyze** > **Run Code Analysis and Suppress Active Issues**.

> **NOTE**
>
> If you migrate a project to Visual Studio 2019, you might suddenly be faced with a large number of code analysis warnings. If you aren't ready to fix the warnings, you can suppress all of them by choosing **Analyze** > **Build and Suppress Active Issues**.

SuppressMessage attribute

When you choose **Suppress** from the context or right-click menu of a code analysis warning in the **Error List**, a SuppressMessageAttribute attribute is added either in your code or to the project's global suppression file.

The SuppressMessageAttribute attribute has the following format:

```
<Scope:SuppressMessage("Rule Category", "Rule Id", Justification = "Justification", MessageId = "MessageId",
Scope = "Scope", Target = "Target")>
```

```
[Scope:SuppressMessage("Rule Category", "Rule Id", Justification = "Justification", MessageId = "MessageId",
Scope = "Scope", Target = "Target")]
```

```
CA_SUPPRESS_MESSAGE("Rule Category", "Rule Id", Justification = "Justification", MessageId = "MessageId",
Scope = "Scope", Target = "Target")
```

The properties of the attribute include:

- **Category** - The category in which the rule is defined. For more information about code analysis rule categories, see Managed code warnings.
- **CheckId** - The identifier of the rule. Support includes both a short and long name for the rule identifier. The short name is CAXXXX; the long name is CAXXXX:FriendlyTypeName.
- **Justification** - The text that is used to document the reason for suppressing the message.
- **MessageId** - Unique identifier of a problem for each message.
- **Scope** - The target on which the warning is being suppressed. If the target is not specified, it is set to the target of the attribute. Supported scopes include the following:
  - `module` - This scope suppresses warnings against an assembly. It is a global suppression that applies to the entire project.
  - `resource` - (legacy FxCop only) This scope suppresses warnings in diagnostic info written to resource files that are part of the module (assembly). This scope is not read/respected in C#/VB compilers for Roslyn analyzer diagnostics, which only analyzes source files.
  - `type` - This scope suppresses warnings against a type.
  - `member` - This scope suppresses warnings against a member.
  - `namespace` - This scope suppresses warnings against the namespace itself. It does not suppress warnings against types within the namespace.
  - `namespaceanddescendants` - (Requires compiler version 3.x or higher and Visual Studio 2019) This scope suppresses warnings in a namespace and all its descendant symbols. The `namespaceanddescendants` value is ignored by legacy analysis.
- **Target** - An identifier that is used to specify the target on which the warning is being suppressed. It must contain a fully qualified item name.

When you see warnings in Visual Studio, you can view examples of `SuppressMessage` by adding a suppression to the global suppression file. The suppression attribute and its required properties appear in a preview window.

SuppressMessage usage

Code Analysis warnings are suppressed at the level to which the SuppressMessageAttribute attribute is applied. For example, the attribute can be applied at the assembly, module, type, member, or parameter level. The purpose of this is to tightly couple the suppression information to the code where the violation occurs.

The general form of suppression includes the rule category and a rule identifier, which contains an optional human-readable representation of the rule name. For example:

```
[SuppressMessage("Microsoft.Design", "CA1039:ListsAreStrongTyped")]
```

If there are strict performance reasons for minimizing in-source suppression metadata, the rule name can be omitted. The rule category and its rule ID together constitute a sufficiently unique rule identifier. For example:

```
[SuppressMessage("Microsoft.Design", "CA1039")]
```

For maintainability reasons, omitting the rule name is not recommended.

Suppress selective violations within a method body

Suppression attributes can be applied to a method, but cannot be embedded within a method body. This means that all violations of a particular rule are suppressed if you add the SuppressMessageAttribute attribute to the method.

In some cases, you might want to suppress a particular instance of the violation, for example so that future code isn't automatically exempt from the code analysis rule. Certain code analysis rules allow you to do this by using the `MessageId` property of the SuppressMessageAttribute attribute. In general, legacy rules for violations on a particular symbol (a local variable or parameter) respect the `MessageId` property. CA1500:VariableNamesShouldNotMatchFieldNames is an example of such a rule. However, legacy rules for violations on executable code (non-symbol) do not respect the `MessageId` property. Additionally, .NET Compiler Platform ("Roslyn") analyzers do not respect the `MessageId` property.

To suppress a particular symbol violation of a rule, specify the symbol name for the `MessageId` property of the SuppressMessageAttribute attribute. The following example shows code with two violations of CA1500:VariableNamesShouldNotMatchFieldNames—one for the `name` variable and one for the `age` variable. Only the violation for the `age` symbol is suppressed.

```
Public Class Animal
    Dim age As Integer
    Dim name As String

    <CodeAnalysis.SuppressMessage("Microsoft.Maintainability", "CA1500:VariableNamesShouldNotMatchFieldNames",
MessageId:="age")>
    Sub PrintInfo()
        Dim age As Integer = 5
        Dim name As String = "Charlie"

        Console.WriteLine("Age {0}, Name {1}", age, name)
    End Sub

End Class
```

```
public class Animal
{
    int age;
    string name;

    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Maintainability",
"CA1500:VariableNamesShouldNotMatchFieldNames", MessageId = "age")]
    private void PrintInfo()
    {
        int age = 5;
        string name = "Charlie";

        Console.WriteLine($"Age {age}, Name {name}");
    }
}
```

Global-level suppressions

The managed code analysis tool examines `SuppressMessage` attributes that are applied at the assembly, module, type, member, or parameter level. It also fires violations against resources and namespaces. These violations must be applied at the global level and are scoped and targeted. For example, the following message suppresses a namespace violation:

```
[module: SuppressMessage("Microsoft.Design", "CA1020:AvoidNamespacesWithFewTypes", Scope = "namespace", Target
= "MyNamespace")]
```

> **NOTE**
>
> When you suppress a warning with `namespace` scope, it suppresses the warning against the namespace itself. It does not suppress the warning against types within the namespace.

Any suppression can be expressed by specifying an explicit scope. These suppressions must live at the global level. You cannot specify member-level suppression by decorating a type.

Global-level suppressions are the only way to suppress messages that refer to compiler-generated code that does not map to explicitly provided user source. For example, the following code suppresses a violation against a compiler-emitted constructor:

```
[module: SuppressMessage("Microsoft.Design", "CA1055:AbstractTypesDoNotHavePublicConstructors", Scope="member",
Target="Microsoft.Tools.FxCop.Type..ctor()")]
```

> **NOTE**
>
> `Target` always contains the fully qualified item name.

Global suppression file

The global suppression file maintains suppressions that are either global-level suppressions or suppressions that do not specify a target. For example, suppressions for assembly-level violations are stored in this file. Additionally, some ASP.NET suppressions are stored in this file because project-level settings are not available for code behind a form. A global suppression file is created and added to your project the first time that you select the **In Project Suppression File** option of the **Suppress** command in the **Error List** window.

Module suppression scope

You can suppress code quality violations for the entire assembly by using the **module** scope.

For example, the following attribute in your *GlobalSuppressions* project file will suppress the ConfigureAwait violation for an ASP.NET Core project:

```
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage("Reliability", "CA2007:Consider calling
ConfigureAwait on the awaited task", Justification = "ASP.NET Core doesn't use thread context to store request
context.", Scope = "module")]
```

Generated code

Managed code compilers and some third-party tools generate code to facilitate rapid code development. Compiler-generated code that appears in source files is usually marked with the `GeneratedCodeAttribute` attribute.

For source code analysis (FxCop analyzers), you can suppress messages in generated code using the .editorconfig file in the root of your project or solution. Use a file pattern to match the generated code. For example, to exclude CS1591 warnings in **.designer.cs* files, use this in the configuration file.

```
[*.designer.cs]
dotnet_diagnostic.CS1591.severity = none
```

For legacy code analysis, you can choose whether to suppress code analysis warnings and errors for generated code. For information about how to suppress such warnings and errors, see How to: Suppress Warnings for Generated Code.

> **NOTE**
>
> Code analysis ignores `GeneratedCodeAttribute` when it is applied to either an entire assembly or a single parameter.

See also

- Scope
- System.Diagnostics.CodeAnalysis
- Use code analyzers

Code Analysis for Managed Code Warnings

8/4/2020 • 2 minutes to read • Edit Online

The Managed Code Analysis tool provides warnings that indicate rule violations in managed code libraries. The warnings are organized into rule areas such as design, localization, performance, and security. Each warning signifies a violation of a Managed Code Analysis rule. This section provides in-depth discussions and examples for each Managed Code Analysis warning.

The following table shows the type of information that is provided for each warning.

ITEM	DESCRIPTION
Type	The TypeName for the rule.
CheckId	The unique identifier for the rule. CheckId and Category are used for in-source suppression of a warning.
Category	The category of the warning.
Breaking change	Whether the fix for a violation of the rule is a breaking change. Breaking change means that an assembly that has a dependency on the target that caused the violation will not recompile with the new fixed version or might fail at run time because of the change. When multiple fixes are available and at least one fix is a breaking change and one fix is not, both 'Breaking' and 'Non-breaking' are specified.
Cause	The specific managed code that causes the rule to generate a warning.
Description	Discusses the issues that are behind the warning.
How to Fix Violations	Explains how to change the source code to satisfy the rule and prevent it from generating a warning.
When to Suppress Warnings	Describes when it is safe to suppress a warning from the rule.
Example Code	Examples that violate the rule and corrected examples that satisfy the rule.
Related Warnings	Related warnings.

In This Section

CATEGORY	DESCRIPTION
Warnings By CheckId	Lists all warnings by CheckId
Cryptography Warnings	Warnings that support safer libraries and applications through the correct use of cryptography.

CATEGORY	DESCRIPTION
Design Warnings	Warnings that support correct library design as specified by the .NET Design Guidelines.
Documentation Warnings	Warnings that support well-documented library design through the correct use of XML documentation comments.
Globalization Warnings	Warnings that support world-ready libraries and applications.
Interoperability Warnings	Warnings that support interaction with COM clients.
Maintainability Warnings	Warnings that support library and application maintenance.
Mobility Warnings	Warnings that support efficient power usage.
Naming Warnings	Warnings that support adherence to the naming conventions of the .NET Design Guidelines.
Performance Warnings	Warnings that support high-performance libraries and applications.
Portability Warnings	Warnings that support portability across different platforms.
Reliability Warnings	Warnings that support library and application reliability, such as correct memory and thread usage.
Security Warnings	Warnings that support safer libraries and applications.
Usage Warnings	Warnings that support appropriate usage of .NET.
Code Analysis Policy Errors	Errors that occur if the code analysis policy is not satisfied at check-in.

Code analysis warnings for managed code by CheckId

8/4/2020 • 62 minutes to read • Edit Online

The following table lists Code Analysis warnings for managed code by the CheckId identifier of the warning.

CHECKID	WARNING	DESCRIPTION
CA1000	CA1000: Do not declare static members on generic types	When a static member of a generic type is called, the type argument must be specified for the type. When a generic instance member that does not support inference is called, the type argument must be specified for the member. In these two cases, the syntax for specifying the type argument is different and easily confused.
CA1001	CA1001: Types that own disposable fields should be disposable	A class declares and implements an instance field that is a System.IDisposable type, and the class does not implement IDisposable. A class that declares an IDisposable field indirectly owns an unmanaged resource and should implement the IDisposable interface.
CA1002	CA1002: Do not expose generic lists	System.Collections.Generic.List<(Of <(T>)>) is a generic collection that is designed for performance, not inheritance. Therefore, List does not contain any virtual members. The generic collections that are designed for inheritance should be exposed instead.
CA1003	CA1003: Use generic event handler instances	A type contains a delegate that returns void, whose signature contains two parameters (the first an object and the second a type that is assignable to EventArgs), and the containing assembly targets Microsoft .NET Framework 2.0.

CHECKID	WARNING	DESCRIPTION
CA1004	CA1004: Generic methods should provide type parameter	Inference is how the type argument of a generic method is determined by the type of argument that is passed to the method, instead of by the explicit specification of the type argument. To enable inference, the parameter signature of a generic method must include a parameter that is of the same type as the type parameter for the method. In this case, the type argument does not have to be specified. When using inference for all type parameters, the syntax for calling generic and non-generic instance methods is identical; this simplifies the usability of generic methods.
CA1005	CA1005: Avoid excessive parameters on generic types	The more type parameters a generic type contains, the more difficult it is to know and remember what each type parameter represents. It is usually obvious with one type parameter, as in List<T>, and in certain cases that have two type parameters, as in Dictionary<TKey, TValue>. However, if more than two type parameters exist, the difficulty becomes too great for most users.
CA1006	CA1006: Do not nest generic types in member signatures	A nested type argument is a type argument that is also a generic type. To call a member whose signature contains a nested type argument, the user must instantiate one generic type and pass this type to the constructor of a second generic type. The required procedure and syntax are complex and should be avoided.
CA1007	CA1007: Use generics where appropriate	An externally visible method contains a reference parameter of type System.Object. Use of a generic method enables all types, subject to constraints, to be passed to the method without first casting the type to the reference parameter type.
CA1008	CA1008: Enums should have zero value	The default value of an uninitialized enumeration, just as other value types, is zero. A nonflags-attributed enumeration should define a member by using the value of zero so that the default value is a valid value of the enumeration. If an enumeration that has the FlagsAttribute attribute applied defines a zero-valued member, its name should be "None" to indicate that no values have been set in the enumeration.

CHECKID	WARNING	DESCRIPTION
CA1009	CA1009: Declare event handlers correctly	Event handler methods take two parameters. The first is of type System.Object and is named "sender". This is the object that raised the event. The second parameter is of type System.EventArgs and is named "e". This is the data that is associated with the event. Event handler methods should not return a value; in the C# programming language, this is indicated by the return type void.
CA1010	CA1010: Collections should implement generic interface	To broaden the usability of a collection, implement one of the generic collection interfaces. Then the collection can be used to populate generic collection types.
CA1011	CA1011: Consider passing base types as parameters	When a base type is specified as a parameter in a method declaration, any type that is derived from the base type can be passed as the corresponding argument to the method. If the additional functionality that is provided by the derived parameter type is not required, use of the base type enables wider use of the method.
CA1012	CA1012: Abstract types should not have constructors	Constructors on abstract types can be called only by derived types. Because public constructors create instances of a type, and you cannot create instances of an abstract type, an abstract type that has a public constructor is incorrectly designed.
CA1013	CA1013: Overload operator equals on overloading add and subtract	A public or protected type implements the addition or subtraction operators without implementing the equality operator.
CA1014	CA1014: Mark assemblies with CLSCompliantAttribute	The Common Language Specification (CLS) defines naming restrictions, data types, and rules to which assemblies must conform if they will be used across programming languages. Good design dictates that all assemblies explicitly indicate CLS compliance by using CLSCompliantAttribute . If this attribute is not present on an assembly, the assembly is not compliant.

CHECKID	WARNING	DESCRIPTION
CA1016	CA1016: Mark assemblies with AssemblyVersionAttribute	.NET uses the version number to uniquely identify an assembly and to bind to types in strongly named assemblies. The version number is used together with version and publisher policy. By default, applications run only with the assembly version with which they were built.
CA1017	CA1017: Mark assemblies with ComVisibleAttribute	ComVisibleAttribute determines how COM clients access managed code. Good design dictates that assemblies explicitly indicate COM visibility. COM visibility can be set for the whole assembly and then overridden for individual types and type members. If this attribute is not present, the contents of the assembly are visible to COM clients.
CA1018	CA1018: Mark attributes with AttributeUsageAttribute	When you define a custom attribute, mark it by using AttributeUsageAttribute to indicate where in the source code the custom attribute can be applied. The meaning and intended usage of an attribute will determine its valid locations in code.
CA1019	CA1019: Define accessors for attribute arguments	Attributes can define mandatory arguments that must be specified when you apply the attribute to a target. These are also known as positional arguments because they are supplied to attribute constructors as positional parameters. For every mandatory argument, the attribute should also provide a corresponding read-only property so that the value of the argument can be retrieved at execution time. Attributes can also define optional arguments, which are also known as named arguments. These arguments are supplied to attribute constructors by name and should have a corresponding read/write property.
CA1020	CA1020: Avoid namespaces with few types	Make sure that each of your namespaces has a logical organization, and that a valid reason exists for putting types in a sparsely populated namespace.
CA1021	CA1021: Avoid out parameters	Passing types by reference (using out or ref) requires experience with pointers, understanding how value types and reference types differ, and handling methods with multiple return values. Also, the difference between out and ref parameters is not widely understood.

CHECKID	WARNING	DESCRIPTION
CA1023	CA1023: Indexers should not be multidimensional	Indexers (that is, indexed properties) should use a single index. Multidimensional indexers can significantly reduce the usability of the library.
CA1024	CA1024: Use properties where appropriate	A public or protected method has a name that starts with "Get", takes no parameters, and returns a value that is not an array. The method might be a good candidate to become a property.
CA1025	CA1025: Replace repetitive arguments with params array	Use a parameter array instead of repeated arguments when the exact number of arguments is unknown and when the variable arguments are the same type or can be passed as the same type.
CA1026	CA1026: Default parameters should not be used	Methods that use default parameters are allowed under the CLS; however, the CLS lets compilers ignore the values that are assigned to these parameters. To maintain the behavior that you want across programming languages, methods that use default parameters should be replaced by method overloads that provide the default parameters.
CA1027	CA1027: Mark enums with FlagsAttribute	An enumeration is a value type that defines a set of related named constants. Apply FlagsAttribute to an enumeration when its named constants can be meaningfully combined.
CA1028	CA1028: Enum storage should be Int32	An enumeration is a value type that defines a set of related named constants. By default, the System.Int32 data type is used to store the constant value. Although you can change this underlying type, it is not required or recommended for most scenarios.
CA1030	CA1030: Use events where appropriate	This rule detects methods that have names that ordinarily would be used for events. If a method is called in response to a clearly defined state change, the method should be invoked by an event handler. Objects that call the method should raise events instead of calling the method directly.
CA1031	CA1031: Do not catch general exception types	General exceptions should not be caught. Catch a more specific exception, or rethrow the general exception as the last statement in the catch block.

CHECKID	WARNING	DESCRIPTION
CA1032	CA1032: Implement standard exception constructors	Failure to provide the full set of constructors can make it difficult to correctly handle exceptions.
CA1033	CA1033: Interface methods should be callable by child types	An unsealed externally visible type provides an explicit method implementation of a public interface and does not provide an alternative externally visible method that has the same name.
CA1034	CA1034: Nested types should not be visible	A nested type is a type that is declared in the scope of another type. Nested types are useful to encapsulate private implementation details of the containing type. Used for this purpose, nested types should not be externally visible.
CA1035	CA1035: ICollection implementations have strongly typed members	This rule requires ICollection implementations to provide strongly typed members so that users are not required to cast arguments to the Object type when they use the functionality that is provided by the interface. This rule assumes that the type that implements ICollection does so to manage a collection of instances of a type that is stronger than Object.
CA1036	CA1036: Override methods on comparable types	A public or protected type implements the System.IComparable interface. It does not override Object.Equals nor does it overload the language-specific operator for equality, inequality, less than, or greater than.
CA1038	CA1038: Enumerators should be strongly typed	This rule requires IEnumerator implementations to also provide a strongly typed version of the Current property so that users are not required to cast the return value to the strong type when they use the functionality that is provided by the interface.
CA1039	CA1039: Lists are strongly typed	This rule requires IList implementations to provide strongly typed members so that users are not required to cast arguments to the System.Object type when they use the functionality that is provided by the interface.

CHECKID	WARNING	DESCRIPTION
CA1040	CA1040: Avoid empty interfaces	Interfaces define members that provide a behavior or usage contract. The functionality that is described by the interface can be adopted by any type, regardless of where the type appears in the inheritance hierarchy. A type implements an interface by providing implementations for the members of the interface. An empty interface does not define any members; therefore, it does not define a contract that can be implemented.
CA1041	CA1041: Provide ObsoleteAttribute message	A type or member is marked by using a System.ObsoleteAttribute attribute that does not have its ObsoleteAttribute.Message property specified. When a type or member that is marked by using ObsoleteAttribute is compiled, the Message property of the attribute is displayed. This gives the user information about the obsolete type or member.
CA1043	CA1043: Use integral or string argument for indexers	Indexers (that is, indexed properties) should use integral or string types for the index. These types are typically used for indexing data structures and they increase the usability of the library. Use of the Object type should be restricted to those cases where the specific integral or string type cannot be specified at design time.
CA1044	CA1044: Properties should not be write only	Although it is acceptable and often necessary to have a read-only property, the design guidelines prohibit the use of write-only properties. This is because letting a user set a value, and then preventing the user from viewing that value, does not provide any security. Also, without read access, the state of shared objects cannot be viewed, which limits their usefulness.
CA1045	CA1045: Do not pass types by reference	Passing types by reference (using out or ref) requires experience with pointers, understanding how value types and reference types differ, and handling methods that have multiple return values. Library architects who design for a general audience should not expect users to master working with `out` or `ref` parameters.

CHECKID	WARNING	DESCRIPTION
CA1046	CA1046: Do not overload operator equals on reference types	For reference types, the default implementation of the equality operator is almost always correct. By default, two references are equal only if they point to the same object.
CA1047	CA1047: Do not declare protected members in sealed types	Types declare protected members so that inheriting types can access or override the member. By definition, sealed types cannot be inherited, which means that protected methods on sealed types cannot be called.
CA1048	CA1048: Do not declare virtual members in sealed types	Types declare methods as virtual so that inheriting types can override the implementation of the virtual method. By definition, a sealed type cannot be inherited. This makes a virtual method on a sealed type meaningless.
CA1049	CA1049: Types that own native resources should be disposable	Types that allocate unmanaged resources should implement IDisposable to enable callers to release those resources on demand and to shorten the lifetimes of the objects that hold the resources.
CA1050	CA1050: Declare types in namespaces	Types are declared in namespaces to prevent name collisions and as a way to organize related types in an object hierarchy.
CA1051	CA1051: Do not declare visible instance fields	The primary use of a field should be as an implementation detail. Fields should be private or internal and should be exposed by using properties.
CA1052	CA1052: Static holder types should be sealed	A public or protected type contains only static members and is not declared by using the sealed (C# Reference) (NotInheritable) modifier. A type that is not meant to be inherited should be marked by using the sealed modifier to prevent its use as a base type.
CA1053	CA1053: Static holder types should not have constructors	A public or nested public type declares only static members and has a public or protected default constructor. The constructor is unnecessary because calling static members does not require an instance of the type. The string overload should call the uniform resource identifier (URI) overload by using the string argument for safety and security.

CHECKID	WARNING	DESCRIPTION
CA1054	CA1054: URI parameters should not be strings	If a method takes a string representation of a URI, a corresponding overload should be provided that takes an instance of the URI class, which provides these services in a safe and secure manner.
CA1055	CA1055: URI return values should not be strings	This rule assumes that the method returns a URI. A string representation of a URI is prone to parsing and encoding errors, and can lead to security vulnerabilities. The System.Uri class provides these services in a safe and secure manner.
CA1056	CA1056: URI properties should not be strings	This rule assumes that the property represents a Uniform Resource Identifier (URI). A string representation of a URI is prone to parsing and encoding errors, and can lead to security vulnerabilities. The System.Uri class provides these services in a safe and secure manner.
CA1057	CA1057: String URI overloads call System.Uri overloads	A type declares method overloads that differ only by the replacement of a string parameter with a System.Uri parameter. The overload that takes the string parameter does not call the overload that takes the URI parameter.
CA1058	CA1058: Types should not extend certain base types	An externally visible type extends certain base types. Use one of the alternatives.
CA1059	CA1059: Members should not expose certain concrete types	A concrete type is a type that has a complete implementation and therefore can be instantiated. To enable widespread use of the member, replace the concrete type by using the suggested interface.
CA1060	CA1060: Move P/Invokes to NativeMethods class	Platform Invocation methods, such as those that are marked by using the System.Runtime.InteropServices.DllImportAttribute attribute, or methods that are defined by using the Declare keyword in Visual Basic, access unmanaged code. These methods should be of the NativeMethods, SafeNativeMethods, or UnsafeNativeMethods class.

CHECKID	WARNING	DESCRIPTION
CA1061	CA1061: Do not hide base class methods	A method in a base type is hidden by an identically named method in a derived type, when the parameter signature of the derived method differs only by types that are more weakly derived than the corresponding types in the parameter signature of the base method.
CA1062	CA1062: Validate arguments of public methods	All reference arguments that are passed to externally visible methods should be checked against null.
CA1063	CA1063: Implement IDisposable correctly	All IDisposable types should implement the Dispose pattern correctly.
CA1064	CA1064: Exceptions should be public	An internal exception is visible only inside its own internal scope. After the exception falls outside the internal scope, only the base exception can be used to catch the exception. If the internal exception is inherited from Exception, SystemException, or ApplicationException, the external code will not have sufficient information to know what to do with the exception.
CA1065	CA1065: Do not raise exceptions in unexpected locations	A method that is not expected to throw exceptions throws an exception.
CA1066	CA1066: Implement IEquatable when overriding Equals	A value type overrides Equals method, but does not implement IEquatable<T>.
CA1067	CA1067: Override Equals when implementing IEquatable	A type implements IEquatable<T>, but does not override Equals method.
CA1068	CA1068: CancellationToken parameters must come last	A method has a CancellationToken parameter that is not the last parameter.
CA1069	CA1069: Enums should not have duplicate values	An enumeration has multiple members which are explicitly assigned the same constant value.
CA1070	CA1070: Do not declare event fields as virtual	A field-like event was declared as virtual.

CHECKID	WARNING	DESCRIPTION
CA1200	CA1200: Avoid using cref tags with a prefix	The cref attribute in an XML documentation tag means "code reference". It specifies that the inner text of the tag is a code element, such as a type, method, or property. Avoid using `cref` tags with prefixes, because it prevents the compiler from verifying references. It also prevents the Visual Studio integrated development environment (IDE) from finding and updating these symbol references during refactorings.
CA1300	CA1300: Specify MessageBoxOptions	To correctly display a message box for cultures that use a right-to-left reading order, the RightAlign and RtlReading members of the MessageBoxOptions enumeration must be passed to the Show method.
CA1301	CA1301: Avoid duplicate accelerators	An access key, also known as an accelerator, enables keyboard access to a control by using the ALT key. When multiple controls have duplicate access keys, the behavior of the access key is not well-defined.
CA1302	CA1302: Do not hardcode locale specific strings	The System.Environment.SpecialFolder enumeration contains members that refer to special system folders. The locations of these folders can have different values on different operating systems; the user can change some of the locations; and the locations are localized. The Environment.GetFolderPath method returns the locations that are associated with the Environment.SpecialFolder enumeration, localized and appropriate for the currently running computer.
CA1303	CA1303: Do not pass literals as localized parameters	An externally visible method passes a string literal as a parameter to a .NET constructor or method, and that string should be localizable.
CA1304	CA1304: Specify CultureInfo	A method or constructor calls a member that has an overload that accepts a System.Globalization.CultureInfo parameter, and the method or constructor does not call the overload that takes the CultureInfo parameter. When a CultureInfo or System.IFormatProvider object is not supplied, the default value that is supplied by the overloaded member might not have the effect that you want in all locales.

CHECKID	WARNING	DESCRIPTION
CA1305	CA1305: Specify IFormatProvider	A method or constructor calls one or more members that have overloads that accept a System.IFormatProvider parameter, and the method or constructor does not call the overload that takes the IFormatProvider parameter. When a System.Globalization.CultureInfo or IFormatProvider object is not supplied, the default value that is supplied by the overloaded member might not have the effect that you want in all locales.
CA1306	CA1306: Set locale for data types	The locale determines culture-specific presentation elements for data, such as formatting that is used for numeric values, currency symbols, and sort order. When you create a DataTable or DataSet, you should explicitly set the locale.
CA1307	CA1307: Specify StringComparison	A string comparison operation uses a method overload that does not set a StringComparison parameter.
CA1308	CA1308: Normalize strings to uppercase	Strings should be normalized to uppercase. A small group of characters cannot make a round trip when they are converted to lowercase.
CA1309	CA1309: Use ordinal StringComparison	A string comparison operation that is nonlinguistic does not set the StringComparison parameter to either Ordinal or OrdinalIgnoreCase. By explicitly setting the parameter to either StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase, your code often gains speed, becomes more correct, and becomes more reliable.
CA1400	CA1400: P/Invoke entry points should exist	A public or protected method is marked by using the System.Runtime.InteropServices.DllImportAttribute attribute. Either the unmanaged library could not be located or the method could not be matched to a function in the library.
CA1401	CA1401: P/Invokes should not be visible	A public or protected method in a public type has the System.Runtime.InteropServices.DllImportAttribute attribute (also implemented by the Declare keyword in Visual Basic). Such methods should not be exposed.

CHECKID	WARNING	DESCRIPTION
CA1402	CA1402: Avoid overloads in COM visible interfaces	When overloaded methods are exposed to COM clients, only the first method overload retains its name. Subsequent overloads are uniquely renamed by appending to the name an underscore character (_) and an integer that corresponds to the order of declaration of the overload.
CA1403	CA1403: Auto layout types should not be COM visible	A COM-visible value type is marked by using the System.Runtime.InteropServices.StructLayoutAttribute attribute set to LayoutKind.Auto. The layout of these types can change between versions of .NET, which will break COM clients that expect a specific layout.
CA1404	CA1404: Call GetLastError immediately after P/Invoke	A call is made to the Marshal.GetLastWin32Error method or the equivalent Win32 GetLastError function, and the immediately previous call is not to an operating system invoke method.
CA1405	CA1405: COM visible type base types should be COM visible	A COM-visible type derives from a type that is not COM-visible.
CA1406	CA1406: Avoid Int64 arguments for Visual Basic 6 clients	Visual Basic 6 COM clients cannot access 64-bit integers.
CA1407	CA1407: Avoid static members in COM visible types	COM does not support static methods.
CA1408	CA1408: Do not use AutoDual ClassInterfaceType	Types that use a dual interface enable clients to bind to a specific interface layout. Any changes in a future version to the layout of the type or any base types will break COM clients that bind to the interface. By default, if the ClassInterfaceAttribute attribute is not specified, a dispatch-only interface is used.
CA1409	CA1409: Com visible types should be creatable	A reference type that is specifically marked as visible to COM contains a public parameterized constructor but does not contain a public default (parameterless) constructor. A type without a public default constructor is not creatable by COM clients.

CHECKID	WARNING	DESCRIPTION
CA1410	CA1410: COM registration methods should be matched	A type declares a method that is marked by using the System.Runtime.InteropServices.ComRegisterFunctionAttribute attribute but does not declare a method marked by using the System.Runtime.InteropServices.ComUnregisterFunctionAttribute attribute, or vice versa.
CA1411	CA1411: COM registration methods should not be visible	A method marked by using the System.Runtime.InteropServices.ComRegisterFunctionAttribute attribute or the System.Runtime.InteropServices.ComUnregisterFunctionAttribute attribute is externally visible.
CA1412	CA1412: Mark ComSource Interfaces as IDispatch	A type is marked by using the System.Runtime.InteropServices.ComSourceInterfacesAttribute attribute, and at least one of the specified interfaces is not marked by using the System.Runtime.InteropServices.InterfaceTypeAttribute attribute set to ComInterfaceType.InterfaceIsIDispatch.
CA1413	CA1413: Avoid non-public fields in COM visible value types	Nonpublic instance fields of COM-visible value types are visible to COM clients. Review the content of the fields for information that should not be exposed, or that will have unintended design or security effects.
CA1414	CA1414: Mark boolean P/Invoke arguments with MarshalAs	The Boolean data type has multiple representations in unmanaged code.
CA1415	CA1415: Declare P/Invokes correctly	This rule looks for operating system invoke method declarations that target Win32 functions that have a pointer to an OVERLAPPED structure parameter and the corresponding managed parameter is not a pointer to a System.Threading.NativeOverlapped structure.
CA1417	CA1417: Do not use `OutAttribute` on string parameters for P/Invokes	String parameters passed by value with the `OutAttribute` can destabilize the runtime if the string is an interned string.
CA1500	CA1500: Variable names should not match field names	An instance method declares a parameter or a local variable whose name matches an instance field of the declaring type, leading to errors.

CHECKID	WARNING	DESCRIPTION
CA1501	CA1501: Avoid excessive inheritance	A type is more than four levels deep in its inheritance hierarchy. Deeply nested type hierarchies can be difficult to follow, understand, and maintain.
CA1502	CA1502: Avoid excessive complexity	This rule measures the number of linearly independent paths through the method, which is determined by the number and complexity of conditional branches.
CA1504	CA1504: Review misleading field names	The name of an instance field starts with "s_", or the name of a static (Shared in Visual Basic) field starts with "m_".
CA1505	CA1505: Avoid unmaintainable code	A type or method has a low maintainability index value. A low maintainability index indicates that a type or method is probably difficult to maintain and would be a good candidate for redesign.
CA1506	CA1506: Avoid excessive class coupling	This rule measures class coupling by counting the number of unique type references that a type or method contains.
CA1507	CA1507: Use nameof in place of string	A string literal is used as an argument where a `nameof` expression could be used.
CA1508	CA1508: Avoid dead conditional code	A method has conditional code that always evaluates to `true` or `false` at runtime. This leads to dead code in the `false` branch of the condition.
CA1509	CA1509: Invalid entry in code metrics configuration file	Code metrics rules, such as CA1501, CA1502, CA1505 and CA1506, supplied a configuration file named `CodeMetricsConfig.txt` that has an invalid entry.
CA1600	CA1600: Do not use idle process priority	Do not set process priority to Idle. Processes that have System.Diagnostics.ProcessPriorityClass. Idle will occupy the CPU when it would otherwise be idle, and will therefore block standby.
CA1601	CA1601: Do not use timers that prevent power state changes	Higher-frequency periodic activity will keep the CPU busy and interfere with power-saving idle timers that turn off the display and hard disks.

CHECKID	WARNING	DESCRIPTION
CA1700	CA1700: Do not name enum values 'Reserved'	This rule assumes that an enumeration member that has a name that contains "reserved" is not currently used but is a placeholder to be renamed or removed in a future version. Renaming or removing a member is a breaking change.
CA1701	CA1701: Resource string compound words should be cased correctly	Each word in the resource string is split into tokens based on the casing. Each contiguous two-token combination is checked by the Microsoft spelling checker library. If recognized, the word produces a violation of the rule.
CA1702	CA1702: Compound words should be cased correctly	The name of an identifier contains multiple words, and at least one of the words appears to be a compound word that is not cased correctly.
CA1703	CA1703: Resource strings should be spelled correctly	A resource string contains one or more words that are not recognized by the Microsoft spelling checker library.
CA1704	CA1704: Identifiers should be spelled correctly	The name of an externally visible identifier contains one or more words that are not recognized by the Microsoft spelling checker library.
CA1707	CA1707: Identifiers should not contain underscores	By convention, identifier names do not contain the underscore (_) character. This rule checks namespaces, types, members, and parameters.
CA1708	CA1708: Identifiers should differ by more than case	Identifiers for namespaces, types, members, and parameters cannot differ only by case because languages that target the common language runtime are not required to be case-sensitive.
CA1709	CA1709: Identifiers should be cased correctly	By convention, parameter names use camel casing and namespace, type, and member names use Pascal casing.
CA1710	CA1710: Identifiers should have correct suffix	By convention, the names of types that extend certain base types or that implement certain interfaces, or types that are derived from these types, have a suffix that is associated with the base type or interface.

CHECKID	WARNING	DESCRIPTION
CA1711	CA1711: Identifiers should not have incorrect suffix	By convention, only the names of types that extend certain base types or that implement certain interfaces, or types that are derived from these types, should end with specific reserved suffixes. Other type names should not use these reserved suffixes.
CA1712	CA1712: Do not prefix enum values with type name	Names of enumeration members are not prefixed by using the type name because development tools are expected to provide type information.
CA1713	CA1713: Events should not have before or after prefix	The name of an event starts with "Before" or "After". To name related events that are raised in a specific sequence, use the present or past tense to indicate the relative position in the sequence of actions.
CA1714	CA1714: Flags enums should have plural names	A public enumeration has the System.FlagsAttribute attribute, and its name does not end in "s". Types that are marked by using FlagsAttribute have names that are plural because the attribute indicates that more than one value can be specified.
CA1715	CA1715: Identifiers should have correct prefix	The name of an externally visible interface does not start with an uppercase "I". The name of a generic type parameter on an externally visible type or method does not start with an uppercase "T".
CA1716	CA1716: Identifiers should not match keywords	A namespace name or a type name matches a reserved keyword in a programming language. Identifiers for namespaces and types should not match keywords that are defined by languages that target the common language runtime.
CA1717	CA1717: Only FlagsAttribute enums should have plural names	Naming conventions dictate that a plural name for an enumeration indicates that more than one value of the enumeration can be specified at the same time.
CA1719	CA1719: Parameter names should not match member names	A parameter name should communicate the meaning of a parameter and a member name should communicate the meaning of a member. It would be a rare design where these were the same. Naming a parameter the same as its member name is unintuitive and makes the library difficult to use.

CHECKID	WARNING	DESCRIPTION
CA1720	CA1720: Identifiers should not contain type names	The name of a parameter in an externally visible member contains a data type name, or the name of an externally visible member contains a language-specific data type name.
CA1721	CA1721: Property names should not match get methods	The name of a public or protected member starts with "Get" and otherwise matches the name of a public or protected property. "Get" methods and properties should have names that clearly distinguish their function.
CA1722	CA1722: Identifiers should not have incorrect prefix	By convention, only certain programming elements have names that begin with a specific prefix.
CA1724	CA1724: Type Names Should Not Match Namespaces	Type names should not match the names of .NET namespaces. Violating this rule can reduce the usability of the library.
CA1725	CA1725: Parameter names should match base declaration	Consistent naming of parameters in an override hierarchy increases the usability of the method overrides. A parameter name in a derived method that differs from the name in the base declaration can cause confusion about whether the method is an override of the base method or a new overload of the method.
CA1726	CA1726: Use preferred terms	The name of an externally visible identifier includes a term for which an alternative, preferred term exists. Alternatively, the name includes the term "Flag" or "Flags".
CA1800	CA1800: Do not cast unnecessarily	Duplicate casts decrease performance, especially when the casts are performed in compact iteration statements.
CA1801	CA1801: Review unused parameters	A method signature includes a parameter that is not used in the method body.
CA1802	CA1802: Use Literals Where Appropriate	A field is declared static and read-only (Shared and ReadOnly in Visual Basic), and is initialized by using a value that is computable at compile time. Because the value that is assigned to the targeted field is computable at compile time, change the declaration to a const (Const in Visual Basic) field so that the value is computed at compile time instead of at run time.

CHECKID	WARNING	DESCRIPTION
CA1804	CA1804: Remove unused locals	Unused local variables and unnecessary assignments increase the size of an assembly and decrease performance.
CA1805	CA1805: Do not initialize unnecessarily	The .NET runtime initializes all fields of reference types to their default values before running the constructor. In most cases, explicitly initializing a field to its default value is redundant, which adds to maintenance costs and may degrade performance (such as with increased assembly size).
CA1806	CA1806: Do not ignore method results	A new object is created but never used; or a method that creates and returns a new string is called and the new string is never used; or a COM or P/Invoke method returns an HRESULT or error code that is never used.
CA1809	CA1809: Avoid excessive locals	A common performance optimization is to store a value in a processor register instead of memory, which is referred to as "enregistering the value". To increase the chance that all local variables are enregistered, limit the number of local variables to 64.
CA1810	CA1810: Initialize reference type static fields inline	When a type declares an explicit static constructor, the just-in-time (JIT) compiler adds a check to each static method and instance constructor of the type to make sure that the static constructor was previously called. Static constructor checks can decrease performance.
CA1811	CA1811: Avoid uncalled private code	A private or internal (assembly-level) member does not have callers in the assembly; it is not invoked by the common language runtime; and it is not invoked by a delegate.
CA1812	CA1812: Avoid uninstantiated internal classes	An instance of an assembly-level type is not created by code in the assembly.
CA1813	CA1813: Avoid unsealed attributes	.NET provides methods for retrieving custom attributes. By default, these methods search the attribute inheritance hierarchy. Sealing the attribute eliminates the search through the inheritance hierarchy and can improve performance.

CHECKID	WARNING	DESCRIPTION
CA1814	CA1814: Prefer jagged arrays over multidimensional	A jagged array is an array whose elements are arrays. The arrays that make up the elements can be of different sizes, leading to less wasted space for some sets of data.
CA1815	CA1815: Override equals and operator equals on value types	For value types, the inherited implementation of Equals uses the Reflection library and compares the contents of all fields. Reflection is computationally expensive, and comparing every field for equality might be unnecessary. If you expect users to compare or sort instances, or to use instances as hash table keys, your value type should implement Equals.
CA1816	CA1816: Call GC.SuppressFinalize correctly	A method that is an implementation of Dispose does not call GC.SuppressFinalize; or a method that is not an implementation of Dispose calls GC.SuppressFinalize; or a method calls GC.SuppressFinalize and passes something other than this (Me in Visual Basic).
CA1819	CA1819: Properties should not return arrays	Arrays that are returned by properties are not write-protected, even when the property is read-only. To keep the array tamper-proof, the property must return a copy of the array. Typically, users will not understand the adverse performance implications of calling such a property.
CA1820	CA1820: Test for empty strings using string length	Comparing strings by using the String.Length property or the String.IsNullOrEmpty method is significantly faster than using Equals.
CA1821	CA1821: Remove empty finalizers	Whenever you can, avoid finalizers because of the additional performance overhead that is involved in tracking object lifetime. An empty finalizer incurs added overhead and delivers no benefit.
CA1822	CA1822: Mark members as static	Members that do not access instance data or call instance methods can be marked as static (Shared in Visual Basic). After you mark the methods as static, the compiler will emit nonvirtual call sites to these members. This can give you a measurable performance gain for performance-sensitive code.
CA1823	CA1823: Avoid unused private fields	Private fields were detected that do not appear to be accessed in the assembly.

CHECKID	WARNING	DESCRIPTION
CA1824	CA1824: Mark assemblies with NeutralResourcesLanguageAttribute	The NeutralResourcesLanguage attribute informs the resource manager of the language that was used to display the resources of a neutral culture for an assembly. This improves lookup performance for the first resource that you load and can reduce your working set.
CA1825	CA1825: Avoid zero-length array allocations	Initializing a zero-length array leads to unnecessary memory allocation. Instead, use the statically allocated empty array instance by calling Array.Empty. The memory allocation is shared across all invocations of this method.
CA1826	CA1826: Use property instead of Linq Enumerable method	Enumerable LINQ method was used on a type that supports an equivalent, more efficient property.
CA1827	CA1827: Do not use Count/LongCount when Any can be used	Count or LongCount method was used where Any method would be more efficient.
CA1828	CA1828: Do not use CountAsync/LongCountAsync when AnyAsync can be used	CountAsync or LongCountAsync method was used where AnyAsync method would be more efficient.
CA1829	CA1829: Use Length/Count property instead of Enumerable.Count method	Count LINQ method was used on a type that supports an equivalent, more efficient `Length` or `Count` property.
CA1830	CA1830: Prefer strongly-typed Append and Insert method overloads on StringBuilder	Append and Insert provide overloads for multiple types beyond String. When possible, prefer the strongly-typed overloads over using ToString() and the string-based overload.
CA1831	CA1831: Use AsSpan instead of Range-based indexers for string when appropriate	When using a range-indexer on a string and implicitly assigning the value to ReadOnlySpan<char> type, the method Substring will be used instead of Slice, which produces a copy of requested portion of the string.
CA1832	CA1832: Use AsSpan or AsMemory instead of Range-based indexers for getting ReadOnlySpan or ReadOnlyMemory portion of an array	When using a range-indexer on an array and implicitly assigning the value to a ReadOnlySpan<T> or ReadOnlyMemory<T> type, the method GetSubArray will be used instead of Slice, which produces a copy of requested portion of the array.

CHECKID	WARNING	DESCRIPTION
CA1833	CA1833: Use AsSpan or AsMemory instead of Range-based indexers for getting Span or Memory portion of an array	When using a range-indexer on an array and implicitly assigning the value to a Span<T> or Memory<T> type, the method GetSubArray will be used instead of Slice, which produces a copy of requested portion of the array.
CA1835	CA1835: Prefer the 'Memory'-based overloads for 'ReadAsync' and 'WriteAsync'	'Stream' has a 'ReadAsync' overload that takes a 'Memory<Byte>' as the first argument, and a 'WriteAsync' overload that takes a 'ReadOnlyMemory<Byte>' as the first argument. Prefer calling the memory based overloads, which are more efficient.
CA1836	CA1836: Prefer `IsEmpty` over `Count` when available	Prefer `IsEmpty` property that is more efficient than `Count`, `Length`, Count<TSource> (IEnumerable<TSource>) or LongCount<TSource> (IEnumerable<TSource>) to determine whether the object contains or not any items.
CA1900	CA1900: Value type fields should be portable	This rule checks that structures that are declared by using explicit layout will align correctly when marshaled to unmanaged code on 64-bit operating systems.
CA1901	CA1901: P/Invoke declarations should be portable	This rule evaluates the size of each parameter and the return value of a P/Invoke, and verifies that the size of the parameter is correct when marshaled to unmanaged code on 32-bit and 64-bit operating systems.
CA1903	CA1903: Use only API from targeted framework	A member or type is using a member or type that was introduced in a service pack that was not included together with the targeted framework of the project.
CA2000	CA2000: Dispose objects before losing scope	Because an exceptional event might occur that will prevent the finalizer of an object from running, the object should be explicitly disposed before all references to it are out of scope.
CA2001	CA2001: Avoid calling problematic methods	A member calls a potentially dangerous or problematic method.

CHECKID	WARNING	DESCRIPTION
CA2002	CA2002: Do not lock on objects with weak identity	An object is said to have a weak identity when it can be directly accessed across application domain boundaries. A thread that tries to acquire a lock on an object that has a weak identity can be blocked by a second thread in a different application domain that has a lock on the same object.
CA2003	CA2003: Do not treat fibers as threads	A managed thread is being treated as a Win32 thread.
CA2004	CA2004: Remove calls to GC.KeepAlive	If you convert to SafeHandle usage, remove all calls to GC.KeepAlive (object). In this case, classes should not have to call GC.KeepAlive. This assumes they do not have a finalizer but rely on SafeHandle to finalize the OS handle for them.
CA2006	CA2006: Use SafeHandle to encapsulate native resources	Use of IntPtr in managed code might indicate a potential security and reliability problem. All uses of IntPtr must be reviewed to determine whether use of a SafeHandle, or similar technology, is required in its place.
CA2007	CA2007: Do not directly await a Task	An asynchronous method awaits a Task directly. When an asynchronous method awaits a Task directly, continuation occurs in the same thread that created the task. This behavior can be costly in terms of performance and can result in a deadlock on the UI thread. Consider calling Task.ConfigureAwait(Boolean) to signal your intention for continuation.
CA2009	CA2009: Do not call ToImmutableCollection on an ImmutableCollection value	`ToImmutable` method was unnecessarily called on an immutable collection from System.Collections.Immutable namespace.
CA2011	CA2011: Do not assign property within its setter	A property was accidentally assigned a value within its own set accessor.
CA2012	CA2012: Use ValueTasks correctly	ValueTasks returned from member invocations are intended to be directly awaited. Attempts to consume a ValueTask multiple times or to directly access one's result before it's known to be completed may result in an exception or corruption. Ignoring such a ValueTask is likely an indication of a functional bug and may degrade performance.

CHECKID	WARNING	DESCRIPTION
CA2013	CA2013: Do not use ReferenceEquals with value types	When comparing values using System.Object.ReferenceEquals, if objA and objB are value types, they are boxed before they are passed to the ReferenceEquals method. This means that even if both objA and objB represent the same instance of a value type, the ReferenceEquals method nevertheless returns false.
CA2014	CA2014: Do not use stackalloc in loops.	Stack space allocated by a stackalloc is only released at the end of the current method's invocation. Using it in a loop can result in unbounded stack growth and eventual stack overflow conditions.
CA2015	CA2015: Do not define finalizers for types derived from MemoryManager<T>	Adding a finalizer to a type derived from MemoryManager<T> may permit memory to be freed while it is still in use by a Span<T>.
CA2016	CA2016: Forward the CancellationToken parameter to methods that take one	Forward the `CancellationToken` parameter to methods that take one to ensure the operation cancellation notifications gets properly propagated, or pass in `CancellationToken.None` explicitly to indicate intentionally not propagating the token.
CA2100	CA2100: Review SQL queries for security vulnerabilities	A method sets the System.Data.IDbCommand.CommandText property by using a string that is built from a string argument to the method. This rule assumes that the string argument contains user input. A SQL command string that is built from user input is vulnerable to SQL injection attacks.
CA2101	CA2101: Specify marshaling for P/Invoke string arguments	A platform invoke member allows partially trusted callers, has a string parameter, and does not explicitly marshal the string. This can cause a potential security vulnerability.
CA2102	CA2102: Catch non-CLSCompliant exceptions in general handlers	A member in an assembly that is not marked by using the RuntimeCompatibilityAttribute or is marked RuntimeCompatibility(WrapNonExceptionThrows = false) contains a catch block that handles System.Exception and does not contain an immediately following general catch block.

CHECKID	WARNING	DESCRIPTION
CA2103	CA2103: Review imperative security	A method uses imperative security and might be constructing the permission by using state information or return values that can change as long as the demand is active. Use declarative security whenever possible.
CA2104	CA2104: Do not declare read only mutable reference types	An externally visible type contains an externally visible read-only field that is a mutable reference type. A mutable type is a type whose instance data can be modified.
CA2105	CA2105: Array fields should not be read only	When you apply the read-only (ReadOnly in Visual Basic) modifier to a field that contains an array, the field cannot be changed to reference a different array. However, the elements of the array that are stored in a read-only field can be changed.
CA2106	CA2106: Secure asserts	A method asserts a permission and no security checks are performed on the caller. Asserting a security permission without performing any security checks can leave an exploitable security weakness in your code.
CA2107	CA2107: Review deny and permit only usage	The PermitOnly method and CodeAccessPermission.Deny security actions should be used only by those who have an advanced knowledge of .NET security. Code that uses these security actions should undergo a security review.
CA2108	CA2108: Review declarative security on value types	A public or protected value type is secured by Data Access or Link Demands.
CA2109	CA2109: Review visible event handlers	A public or protected event-handling method was detected. Event-handling methods should not be exposed unless absolutely necessary.
CA2111	CA2111: Pointers should not be visible	A pointer is not private, internal, or read-only. Malicious code can change the value of the pointer, which potentially gives access to arbitrary locations in memory or causes application or system failures.

CHECKID	WARNING	DESCRIPTION
CA2112	CA2112: Secured types should not expose fields	A public or protected type contains public fields and is secured by Link Demands. If code has access to an instance of a type that is secured by a link demand, the code does not have to satisfy the link demand to access the fields of the type.
CA2114	CA2114: Method security should be a superset of type	A method should not have both method-level and type-level declarative security for the same action.
CA2115	CA2115: Call GC.KeepAlive when using native resources	This rule detects errors that might occur because an unmanaged resource is being finalized while it is still being used in unmanaged code.
CA2116	CA2116: APTCA methods should only call APTCA methods	When the APTCA (AllowPartiallyTrustedCallersAttribute) is present on a fully trusted assembly, and the assembly executes code in another assembly that does not allow for partially trusted callers, a security exploit is possible.
CA2117	CA2117: APTCA types should only extend APTCA base types	When the APTCA is present on a fully trusted assembly, and a type in the assembly inherits from a type that does not allow for partially trusted callers, a security exploit is possible.
CA2118	CA2118: Review SuppressUnmanagedCodeSecurityAttribute usage	SuppressUnmanagedCodeSecurityAttribute changes the default security system behavior for members that execute unmanaged code that uses COM interop or operating system invocation. This attribute is primarily used to increase performance; however, the performance gains come together with significant security risks.
CA2119	CA2119: Seal methods that satisfy private interfaces	An inheritable public type provides an overridable method implementation of an internal (Friend in Visual Basic) interface. To fix a violation of this rule, prevent the method from being overridden outside the assembly.
CA2120	CA2120: Secure serialization constructors	This type has a constructor that takes a System.Runtime.Serialization.SerializationInfo object and a System.Runtime.Serialization.StreamingContext object (the signature of the serialization constructor). This constructor is not secured by a security check, but one or more of the regular constructors in the type are secured.

CHECKID	WARNING	DESCRIPTION
CA2121	CA2121: Static constructors should be private	The system calls the static constructor before the first instance of the type is created or any static members are referenced. If a static constructor is not private, it can be called by code other than the system. Depending on the operations that are performed in the constructor, this can cause unexpected behavior.
CA2122	CA2122: Do not indirectly expose methods with link demands	A public or protected member has Link Demands and is called by a member that does not perform any security checks. A link demand checks the permissions of the immediate caller only.
CA2123	CA2123: Override link demands should be identical to base	This rule matches a method to its base method, which is either an interface or a virtual method in another type, and then compares the link demands on each. If this rule is violated, a malicious caller can bypass the link demand just by calling the unsecured method.
CA2124	CA2124: Wrap vulnerable finally clauses in outer try	A public or protected method contains a try/finally block. The finally block appears to reset the security state and is not itself enclosed in a finally block.
CA2126	CA2126: Type link demands require inheritance demands	A public unsealed type is protected by using a link demand and has an overridable method. Neither the type nor the method is protected by using an inheritance demand.
CA2130	CA2130: Security critical constants should be transparent	Transparency enforcement is not enforced for constant values because compilers inline constant values so that no lookup is required at run time. Constant fields should be security transparent so that code reviewers do not assume that transparent code cannot access the constant.

CHECKID	WARNING	DESCRIPTION
CA2131	CA2131: Security critical types may not participate in type equivalence	A type participates in type equivalence and either the type itself, or a member or field of the type, is marked by using the SecurityCriticalAttribute attribute. This rule occurs on any critical types or types that contain critical methods or fields that are participating in type equivalence. When the CLR detects such a type, it does not load it with a TypeLoadException at run time. Typically, this rule is raised only when users implement type equivalence manually instead of in by relying on tlbimp and the compilers to do the type equivalence.
CA2132	CA2132: Default constructors must be at least as critical as base type default constructors	Types and members that have the SecurityCriticalAttribute cannot be used by Silverlight application code. Security-critical types and members can be used only by trusted code in the .NET Framework for Silverlight class library. Because a public or protected construction in a derived class must have the same or greater transparency than its base class, a class in an application cannot be derived from a class marked as SecurityCritical.
CA2133	CA2133: Delegates must bind to methods with consistent transparency	This warning is raised on a method that binds a delegate that is marked by using the SecurityCriticalAttribute to a method that is transparent or that is marked by using the SecuritySafeCriticalAttribute. The warning also is raised on a method that binds a delegate that is transparent or safe-critical to a critical method.
CA2134	CA2134: Methods must keep consistent transparency when overriding base methods	This rule is raised when a method marked by using the SecurityCriticalAttribute overrides a method that is transparent or marked by using the SecuritySafeCriticalAttribute. The rule also is raised when a method that is transparent or marked by using the SecuritySafeCriticalAttribute overrides a method that is marked by using a SecurityCriticalAttribute. The rule is applied when overriding a virtual method or implementing an interface.
CA2135	CA2135: Level 2 assemblies should not contain LinkDemands	LinkDemands are deprecated in the level 2 security rule set. Instead of using LinkDemands to enforce security at JIT compilation time, mark the methods, types, and fields by using the SecurityCriticalAttribute attribute.

CHECKID	WARNING	DESCRIPTION
CA2127	CA2136: Members should not have conflicting transparency annotations	Critical code cannot occur in a 100 percent-transparent assembly. This rule analyzes 100 percent-transparent assemblies for any SecurityCritical annotations at the type, field, and method levels.
CA2136	CA2136: Members should not have conflicting transparency annotations	Transparency attributes are applied from code elements of larger scope to elements of smaller scope. The transparency attributes of code elements that have larger scope take precedence over transparency attributes of code elements that are contained in the first element. For example, a class that is marked by using the SecurityCriticalAttribute attribute cannot contain a method that is marked by using the SecuritySafeCriticalAttribute attribute.
CA2137	CA2137: Transparent methods must contain only verifiable IL	A method contains unverifiable code or returns a type by reference. This rule is raised on attempts by security transparent code to execute unverifiable microsoft intermediate language (MISL). However, the rule does not contain a full IL verifier, and instead uses heuristics to catch most violations of MSIL verification.
CA2138	CA2138: Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute	A security transparent method calls a method that is marked by using the SuppressUnmanagedCodeSecurityAttribute attribute.
CA2139	CA2139: Transparent methods may not use the HandleProcessCorruptingExceptions attribute	This rule is raised by any method that is transparent and attempts to handle a process corrupting exception by using the HandleProcessCorruptedStateExceptionsAttribute attribute. A process corrupting exception is a CLR version 4.0 exception classification of exceptions such as AccessViolationException. The HandleProcessCorruptedStateExceptionsAttribute attribute may be used only by security critical methods, and will be ignored if it is applied to a transparent method.

CHECKID	WARNING	DESCRIPTION
CA2129	CA2140: Transparent code must not reference security critical items	Methods that are marked by SecurityTransparentAttribute call nonpublic members that are marked as SecurityCritical. This rule analyzes all methods and types in an assembly that is mixed transparent/critical, and flags any calls from transparent code to nonpublic critical code that are not marked as SecurityTreatAsSafe.
CA2140	CA2140: Transparent code must not reference security critical items	A code element that is marked by using the SecurityCriticalAttribute attribute is security critical. A transparent method cannot use a security critical element. If a transparent type attempts to use a security critical type, a TypeAccessException, MethodAccessException, or FieldAccessException is raised.
CA2141	CA2141:Transparent methods must not satisfy LinkDemands	A security transparent method calls a method in an assembly that is not marked by using the APTCA, or a security transparent method satisfies a LinkDemand for a type or a method.
CA2142	CA2142: Transparent code should not be protected with LinkDemands	This rule is raised on transparent methods that require LinkDemands to access them. Security transparent code should not be responsible for verifying the security of an operation, and therefore should not demand permissions.
CA2143	CA2143: Transparent methods should not use security demands	Security transparent code should not be responsible for verifying the security of an operation, and therefore should not demand permissions. Security transparent code should use full demands to make security decisions and safe-critical code should not rely on transparent code to have made the full demand.
CA2144	CA2144: Transparent code should not load assemblies from byte arrays	The security review for transparent code is not as complete as the security review for critical code because transparent code cannot perform security sensitive actions. Assemblies that are loaded from a byte array might not be noticed in transparent code, and that byte array might contain critical, or more important safe-critical code, that does have to be audited.

CHECKID	WARNING	DESCRIPTION
CA2145	CA2145: Transparent methods should not be decorated with the SuppressUnmanagedCodeSecurityAttribute	Methods that are decorated by the SuppressUnmanagedCodeSecurityAttribute attribute have an implicit LinkDemand put upon any method that calls it. This LinkDemand requires that the calling code be security critical. Marking the method that uses SuppressUnmanagedCodeSecurity by using the SecurityCriticalAttribute attribute makes this requirement more obvious for callers of the method.
CA2146	CA2146: Types must be at least as critical as their base types and interfaces	This rule is raised when a derived type has a security transparency attribute that is not as critical as its base type or implemented interface. Only critical types can derive from critical base types or implement critical interfaces, and only critical or safe-critical types can derive from safe-critical base types or implement safe-critical interfaces.
CA2128	CA2147: Transparent methods may not use security asserts	This rule analyzes all methods and types in an assembly that is either 100 percent-transparent or mixed transparent/critical, and flags any declarative or imperative use of Assert.
CA2147	CA2147: Transparent methods may not use security asserts	Code that is marked as SecurityTransparentAttribute is not granted sufficient permissions to assert.
CA2149	CA2149: Transparent methods must not call into native code	This rule is raised on any transparent method that calls directly into native code (for example, through a P/Invoke). Violations of this rule lead to a MethodAccessException in the level 2 transparency model and a full demand for UnmanagedCode in the level 1 transparency model.
CA2151	CA2151: Fields with critical types should be security critical	To use security critical types, the code that references the type must be either security critical or security safe critical. This is true even if the reference is indirect. Therefore, having a security transparent or security safe critical field is misleading because transparent code will still be unable to access the field.
CA2200	CA2200: Rethrow to preserve stack details	An exception is rethrown and the exception is explicitly specified in the throw statement. If an exception is rethrown by specifying the exception in the throw statement, the list of method calls between the original method that threw the exception and the current method is lost.

CHECKID	WARNING	DESCRIPTION
CA2201	CA2201: Do not raise reserved exception types	This makes the original error difficult to detect and debug.
CA2202	CA2202: Do not dispose objects multiple times	A method implementation contains code paths that could cause multiple calls to System.IDisposable.Dispose or a Dispose equivalent (such as a Close() method on some types) on the same object.
CA2204	CA2204: Literals should be spelled correctly	A literal string in a method body contains one or more words that are not recognized by the Microsoft spelling checker library.
CA2205	CA2205: Use managed equivalents of Win32 API	An operating system invoke method is defined and a .NET method that has the equivalent functionality is available.
CA2207	CA2207: Initialize value type static fields inline	A value type declares an explicit static constructor. To fix a violation of this rule, initialize all static data when it is declared and remove the static constructor.
CA2208	CA2208: Instantiate argument exceptions correctly	A call is made to the default (parameterless) constructor of an exception type that is or derives from ArgumentException, or an incorrect string argument is passed to a parameterized constructor of an exception type that is or derives from ArgumentException.
CA2210	CA2210: Assemblies should have valid strong names	The strong name protects clients from unknowingly loading an assembly that has been tampered with. Assemblies without strong names should not be deployed outside very limited scenarios. If you share or distribute assemblies that are not correctly signed, the assembly can be tampered with, the common language runtime might not load the assembly, or the user might have to disable verification on his or her computer.
CA2211	CA2211: Non-constant fields should not be visible	Static fields that are neither constants nor read-only are not thread-safe. Access to such a field must be carefully controlled and requires advanced programming techniques to synchronize access to the class object.

CHECKID	WARNING	DESCRIPTION
CA2212	CA2212: Do not mark serviced components with WebMethod	A method in a type that inherits from System.EnterpriseServices.ServicedComponent is marked by using System.Web.Services.WebMethodAttribute. Because WebMethodAttribute and a ServicedComponent method have conflicting behavior and requirements for context and transaction flow, the behavior of the method will be incorrect in some scenarios.
CA2213	CA2213: Disposable fields should be disposed	A type that implements System.IDisposable declares fields that are of types that also implement IDisposable. The Dispose method of the field is not called by the Dispose method of the declaring type.
CA2214	CA2214: Do not call overridable methods in constructors	When a constructor calls a virtual method, the constructor for the instance that invokes the method may not have executed.
CA2215	CA2215: Dispose methods should call base class dispose	If a type inherits from a disposable type, it must call the Dispose method of the base type from its own Dispose method.
CA2216	CA2216: Disposable types should declare finalizer	A type that implements System.IDisposable and has fields that suggest the use of unmanaged resources does not implement a finalizer, as described by Object.Finalize.
CA2217	CA2217: Do not mark enums with FlagsAttribute	An externally visible enumeration is marked by using FlagsAttribute, and it has one or more values that are not powers of two or a combination of the other defined values on the enumeration.
CA2218	CA2218: Override GetHashCode on overriding Equals	GetHashCode returns a value, based on the current instance, that is suited for hashing algorithms and data structures such as a hash table. Two objects that are the same type and are equal must return the same hash code.
CA2219	CA2219: Do not raise exceptions in exception clauses	When an exception is raised in a finally or fault clause, the new exception hides the active exception. When an exception is raised in a filter clause, the run time silently catches the exception. This makes the original error difficult to detect and debug.

CHECKID	WARNING	DESCRIPTION
CA2220	CA2220: Finalizers should call base class finalizer	Finalization must be propagated through the inheritance hierarchy. To guarantee this, types must call their base class Finalize method in their own Finalize method.
CA2221	CA2221: Finalizers should be protected	Finalizers must use the family access modifier.
CA2222	CA2222: Do not decrease inherited member visibility	You should not change the access modifier for inherited members. Changing an inherited member to private does not prevent callers from accessing the base class implementation of the method.
CA2223	CA2223: Members should differ by more than return type	Although the common language runtime allows the use of return types to differentiate between otherwise identical members, this feature is not in the Common Language Specification, nor is it a common feature of .NET programming languages.
CA2224	CA2224: Override equals on overloading operator equals	A public type implements the equality operator but does not override Object.Equals.
CA2225	CA2225: Operator overloads have named alternates	An operator overload was detected, and the expected named alternative method was not found. The named alternative member provides access to the same functionality as the operator and is provided for developers who program in languages that do not support overloaded operators.
CA2226	CA2226: Operators should have symmetrical overloads	A type implements the equality or inequality operator and does not implement the opposite operator.
CA2227	CA2227: Collection properties should be read only	A writable collection property allows a user to replace the collection with a different collection. A read-only property stops the collection from being replaced but still allows the individual members to be set.
CA2228	CA2228: Do not ship unreleased resource formats	Resource files that were built by using prerelease versions of .NET might not be usable by supported versions of .NET.
CA2229	CA2229: Implement serialization constructors	To fix a violation of this rule, implement the serialization constructor. For a sealed class, make the constructor private; otherwise, make it protected.

CHECKID	WARNING	DESCRIPTION
CA2230	CA2230: Use params for variable arguments	A public or protected type contains a public or protected method that uses the VarArgs calling convention instead of the params keyword.
CA2231	CA2231: Overload operator equals on overriding ValueType.Equals	A value type overrides Object.Equals but does not implement the equality operator.
CA2232	CA2232: Mark Windows Forms entry points with STAThread	STAThreadAttribute indicates that the COM threading model for the application is a single-threaded apartment. This attribute must be present on the entry point of any application that uses Windows Forms; if it is omitted, the Windows components might not work correctly.
CA2233	CA2233: Operations should not overflow	You should not perform arithmetic operations without first validating the operands. This makes sure that the result of the operation is not outside the range of possible values for the data types that are involved.
CA2234	CA2234: Pass System.Uri objects instead of strings	A call is made to a method that has a string parameter whose name contains "uri", "URI", "urn", "URN", "url", or "URL". The declaring type of the method contains a corresponding method overload that has a System.Uri parameter.
CA2235	CA2235: Mark all non-serializable fields	An instance field of a type that is not serializable is declared in a type that is serializable.
CA2236	CA2236: Call base class methods on ISerializable types	To fix a violation of this rule, call the base type GetObjectData method or serialization constructor from the corresponding derived type method or constructor.
CA2237	CA2237: Mark ISerializable types with SerializableAttribute	To be recognized by the common language runtime as serializable, types must be marked by using the SerializableAttribute attribute even when the type uses a custom serialization routine through implementation of the ISerializable interface.
CA2238	CA2238: Implement serialization methods correctly	A method that handles a serialization event does not have the correct signature, return type, or visibility.

CHECKID	WARNING	DESCRIPTION
CA2239	CA2239: Provide deserialization methods for optional fields	A type has a field that is marked by using the System.Runtime.Serialization.OptionalFieldAttribute attribute, and the type does not provide deserialization event handling methods.
CA2240	CA2240: Implement ISerializable correctly	To fix a violation of this rule, make the GetObjectData method visible and overridable, and make sure that all instance fields are included in the serialization process or explicitly marked by using the NonSerializedAttribute attribute.
CA2241	CA2241: Provide correct arguments to formatting methods	The format argument that is passed to System.String.Format does not contain a format item that corresponds to each object argument, or vice versa.
CA2242	CA2242: Test for NaN correctly	This expression tests a value against Single.Nan or Double.Nan. Use Single.IsNan(Single) or Double.IsNan(Double) to test the value.
CA2243	CA2243: Attribute string literals should parse correctly	The string literal parameter of an attribute does not parse correctly for a URL, a GUID, or a version.
CA2244	CA2244: Do not duplicate indexed element initializations	An object initializer has more than one indexed element initializer with the same constant index. All but the last initializer are redundant.
CA2245	CA2245: Do not assign a property to itself	A property was accidentally assigned to itself.
CA2246	CA2246: Do not assign a symbol and its member in the same statement	Assigning a symbol and its member, that is, a field or a property, in the same statement is not recommended. It is not clear if the member access was intended to use the symbol's old value prior to the assignment or the new value from the assignment in this statement.
CA2247	CA2247: Argument passed to TaskCompletionSource constructor should be TaskCreationOptions enum instead of TaskContinuationOptions enum.	TaskCompletionSource has constructors that take TaskCreationOptions that control the underlying Task, and constructors that take object state that's stored in the task. Accidentally passing a TaskContinuationOptions instead of a TaskCreationOptions will result in the call treating the options as state.

CHECKID	WARNING	DESCRIPTION
CA5122	CA5122 P/Invoke declarations should not be safe critical	Methods are marked as SecuritySafeCritical when they perform a security sensitive operation, but are also safe to be used by transparent code. Transparent code may never directly call native code through a P/Invoke. Therefore, marking a P/Invoke as security safe critical will not enable transparent code to call it, and is misleading for security analysis.
CA5359	CA5359 Do not disable certificate validation	A certificate can help authenticate the identity of the server. Clients should validate the server certificate to ensure requests are sent to the intended server. If the ServerCertificateValidationCallback always returns `true`, any certificate will pass validation.
CA5360	CA5360 Do not call dangerous methods in deserialization	Insecure deserialization is a vulnerability which occurs when untrusted data is used to abuse the logic of an application, inflict a Denial-of-Service (DoS) attack, or even execute arbitrary code upon it being deserialized. It's frequently possible for malicious users to abuse these deserialization features when the application is deserializing untrusted data which is under their control. Specifically, invoke dangerous methods in the process of deserialization. Successful insecure deserialization attacks could allow an attacker to carry out attacks such as DoS attacks, authentication bypasses, and remote code execution.
CA5362	CA5362 Potential reference cycle in deserialized object graph	If deserializing untrusted data, then any code processing the deserialized object graph needs to handle reference cycles without going into infinite loops. This includes both code that's part of a deserialization callback and code that processes the object graph after deserialization completed. Otherwise, an attacker could perform a Denial-of-Service attack with malicious data containing a reference cycle.
CA5365	CA5365 Do Not Disable HTTP Header Checking	HTTP header checking enables encoding of the carriage return and newline characters, \r and \n, that are found in response headers. This encoding can help to avoid injection attacks that exploit an application that echoes untrusted data contained by the header.

CHECKID	WARNING	DESCRIPTION
CA5366	CA5366 Use XmlReader For DataSet Read XML	Using a DataSet to read XML with untrusted data may load dangerous external references, which should be restricted by using an XmlReader with a secure resolver or with DTD processing disabled.
CA5367	CA5367 Do Not Serialize Types With Pointer Fields	This rule checks whether there's a serializable class with a pointer field or property. Members that can't be serialized can be a pointer, such as static members or fields marked with NonSerializedAttribute.
CA5368	CA5368 Set ViewStateUserKey For Classes Derived From Page	Setting the ViewStateUserKey property can help you prevent attacks on your application by allowing you to assign an identifier to the view-state variable for individual users so that attackers cannot use the variable to generate an attack. Otherwise, there will be vulnerabilities to cross-site request forgery.
CA5374	CA5374 Do Not Use XslTransform	This rule checks if System.Xml.Xsl.XslTransform is instantiated in the code. System.Xml.Xsl.XslTransform is now obsolete and shouldn't be used.
CA5375	CA5375 Do not use account shared access signature	An account SAS can delegate access to read, write, and delete operations on blob containers, tables, queues, and file shares that are not permitted with a service SAS. However, it doesn't support container-level policies and has less flexibility and control over the permissions that are granted. Once malicious users get it, your storage account will be compromised easily.
CA5376	CA5376 Use SharedAccessProtocol HttpsOnly	SAS is sensitive data that can't be transported in plain text on HTTP.
CA5377	CA5377 Use container level access policy	A container-level access policy can be modified or revoked at any time. It provides greater flexibility and control over the permissions that are granted.
CA5379	CA5379 Do not use weak key derivation function algorithm	The Rfc2898DeriveBytes class defaults to using the SHA1 algorithm. You should specify the hash algorithm to use in some overloads of the constructor with SHA256 or higher. Note, HashAlgorithm property only has a `get` accessor and doesn't have a `overriden` modifier.

CHECKID	WARNING	DESCRIPTION
CA5382	CA5382 Use secure cookies in ASP.NET Core	Applications available over HTTPS must use secure cookies, which indicate to the browser that the cookie should only be transmitted using Secure Sockets Layer (SSL).
CA5383	CA5383 Ensure use secure cookies in ASP.NET Core	Applications available over HTTPS must use secure cookies, which indicate to the browser that the cookie should only be transmitted using Secure Sockets Layer (SSL).
CA5384	CA5384 Do not use digital signature algorithm (DSA)	DSA is a weak asymmetric encryption algorithm.
CA5385	CA5385 Use Rivest–Shamir–Adleman (RSA) algorithm with sufficient key size	An RSA key smaller than 2048 bits is more vulnerable to brute force attacks.
CA5387	CA5387 Do not use weak key derivation function with insufficient iteration count	This rule checks if a cryptographic key was generated by Rfc2898DeriveBytes with an iteration count of less than 100,000. A higher iteration count can help mitigate against dictionary attacks that try to guess the generated cryptographic key.
CA5388	CA5388 Ensure sufficient iteration count when using weak key derivation function	This rule checks if a cryptographic key was generated by Rfc2898DeriveBytes with an iteration count that may be less than 100,000. A higher iteration count can help mitigate against dictionary attacks that try to guess the generated cryptographic key.
CA5390	CA5390 Do not hard-code encryption key	For a symmetric algorithm to be successful, the secret key must be known only to the sender and the receiver. When a key is hard-coded, it is easily discovered. Even with compiled binaries, it is easy for malicious users to extract it. Once the private key is compromised, the cipher text can be decrypted directly and is not protected anymore.
CA5391	CA5391 Use antiforgery tokens in ASP.NET Core MVC controllers	Handling a `POST`, `PUT`, `PATCH`, or `DELETE` request without validating an antiforgery token may be vulnerable to cross-site request forgery attacks. A cross-site request forgery attack can send malicious requests from an authenticated user to your ASP.NET Core MVC controller.

CHECKID	WARNING	DESCRIPTION
CA5392	CA5392 Use DefaultDllImportSearchPaths attribute for P/Invokes	By default, P/Invoke functions using DllImportAttribute probe a number of directories, including the current working directory for the library to load. This can be a security issue for certain applications, leading to DLL hijacking.
CA5393	CA5393 Do not use unsafe DllImportSearchPath value	There could be a malicious DLL in the default DLL search directories and assembly directories. Or, depending on where your application is run from, there could be a malicious DLL in the application's directory.
CA5394	CA5394 Do not use insecure randomness	Using a cryptographically weak pseudo-random number generator may allow an attacker to predict what security-sensitive value will be generated.
CA5395	CA5395 Miss HttpVerb attribute for action methods	All the action methods that create, edit, delete, or otherwise modify data needs to be protected with the antiforgery attribute from cross-site request forgery attacks. Performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.
CA5396	CA5396 Set HttpOnly to true for HttpCookie	As a defense in depth measure, ensure security sensitive HTTP cookies are marked as HttpOnly. This indicates web browsers should disallow scripts from accessing the cookies. Injected malicious scripts are a common way of stealing cookies.
CA5399	CA5399 Definitely disable HttpClient certificate revocation list check	A revoked certificate isn't trusted anymore. It could be used by attackers passing some malicious data or stealing sensitive data in HTTPS communication.
CA5400	CA5400 Ensure HttpClient certificate revocation list check is not disabled	A revoked certificate isn't trusted anymore. It could be used by attackers passing some malicious data or stealing sensitive data in HTTPS communication.
CA5401	CA5401 Do not use CreateEncryptor with non-default IV	Symmetric encryption should always use a non-repeatable initialization vector to prevent dictionary attacks.
CA5402	CA5402 Use CreateEncryptor with the default IV	Symmetric encryption should always use a non-repeatable initialization vector to prevent dictionary attacks.

Cryptography Warnings

1/1/2020 • 2 minutes to read • Edit Online

Cryptography warnings support safer libraries and applications through the correct use of cryptography. These warnings help prevent security flaws in your program. If you disable any of these warnings, you should clearly mark the reason in code and also inform the designated security officer for your development project.

RULE	DESCRIPTION
CA5350: Do Not Use Weak Cryptographic Algorithms	Weak encryption algorithms and hashing functions are used today for a number of reasons, but they should not be used to guarantee the confidentiality or integrity of the data they protect. This rule triggers when it finds TripleDES, SHA1, or RIPEMD160 algorithms in the code.
CA5351 Do Not Use Broken Cryptographic Algorithms	Broken cryptographic algorithms are not considered secure and their use should be strongly discouraged. This rule triggers when it finds the MD5 hash algorithm or either the DES or RC2 encryption algorithms in code.

CA5350: Do Not Use Weak Cryptographic Algorithms

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5350
Category	Microsoft.Cryptography
Breaking change	Non-breaking

NOTE

This warning was last updated on November 2015.

Cause

Encryption algorithms such as TripleDES and hashing algorithms such as SHA1 and RIPEMD160 are considered to be weak.

These cryptographic algorithms do not provide as much security assurance as more modern counterparts. Cryptographic hashing algorithms SHA1 and RIPEMD160 provide less collision resistance than more modern hashing algorithms. The encryption algorithm TripleDES provides fewer bits of security than more modern encryption algorithms.

Rule description

Weak encryption algorithms and hashing functions are used today for a number of reasons, but they should not be used to guarantee the confidentiality of the data they protect.

The rule triggers when it finds 3DES, SHA1 or RIPEMD160 algorithms in the code and throws a warning to the user.

How to fix violations

Use cryptographically stronger options:

- For TripleDES encryption, use Aes encryption.
- For SHA1 or RIPEMD160 hashing functions, use ones in the SHA-2 family (e.g. SHA512, SHA384, SHA256).

When to suppress warnings

Suppress a warning from this rule when the level of protection needed for the data does not require a security guarantee.

Pseudo-code examples

As of the time of this writing, the following pseudo-code sample illustrates the pattern detected by this rule.

SHA-1 Hashing Violation

```
using System.Security.Cryptography;
...
var hashAlg = SHA1.Create();
```

Solution:

```
using System.Security.Cryptography;
...
var hashAlg = SHA256.Create();
```

RIPEMD160 Hashing Violation

```
using System.Security.Cryptography;
...
var hashAlg = RIPEMD160Managed.Create();
```

Solution:

```
using System.Security.Cryptography;
...
var hashAlg = SHA256.Create();
```

TripleDES Encryption Violation

```
using System.Security.Cryptography;
...
using (TripleDES encAlg = TripleDES.Create())
{
  ...
}
```

Solution:

```
using System.Security.Cryptography;
...
using (AesManaged encAlg = new AesManaged())
{
  ...
}
```

CA5351 Do Not Use Broken Cryptographic Algorithms

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5351
Category	Microsoft.Cryptography
Breaking change	Non-breaking

NOTE

This warning was last updated on November 2015.

Cause

Hashing functions such as MD5 and encryption algorithms such as DES and RC2 can expose significant risk and may result in the exposure of sensitive information through trivial attack techniques, such as brute force attacks and hash collisions.

The cryptographic algorithms list below are subject to known cryptographic attacks. The cryptographic hash algorithm MD5 is subject to hash collision attacks. Depending on the usage, a hash collision may lead to impersonation, tampering, or other kinds of attacks on systems that rely on the unique cryptographic output of a hashing function. The encryption algorithms DES and RC2 are subject to cryptographic attacks that may result in unintended disclosure of encrypted data.

Rule description

Broken cryptographic algorithms are not considered secure and their use should be discouraged. The MD5 hash algorithm is susceptible to known collision attacks, though the specific vulnerability will vary based on the context of use. Hashing algorithms used to ensure data integrity (for example, file signature or digital certificate) are particularly vulnerable. In this context, attackers could generate two separate pieces of data, such that benign data can be substituted with malicious data, without changing the hash value or invalidating an associated digital signature.

For encryption algorithms:

- DES encryption contains a small key size, which could be brute-forced in less than a day.
- RC2 encryption is susceptible to a related-key attack, where the attacker finds mathematical relationships between all key values.

This rule triggers when it finds any of the above cryptographic functions in source code and throws a warning to the user.

How to fix violations

Use cryptographically stronger options:

- For MD5, use hashes in the SHA-2 family (for example, SHA512, SHA384, SHA256).
- For DES and RC2, use Aes encryption.

When to suppress warnings

Do not suppress a warning from this rule, unless it's been reviewed by a cryptographic expert.

Pseudo-code Examples

The following pseudo-code samples illustrate the pattern detected by this rule and possible alternatives.

MD5 Hashing Violation

```
using System.Security.Cryptography;
...
var hashAlg = MD5.Create();
```

Solution:

```
using System.Security.Cryptography;
...
var hashAlg = SHA256.Create();
```

RC2 Encryption Violation

```
using System.Security.Cryptography;
...
RC2 encAlg = RC2.Create();
```

Solution:

```
using System.Security.Cryptography;
...
using (AesManaged encAlg = new AesManaged())
{
  ...
}
```

DES Encryption Violation

```
using System.Security.Cryptography;
...
DES encAlg = DES.Create();
```

Solution:

```
using System.Security.Cryptography;
...
using (AesManaged encAlg = new AesManaged())
{
  ...
}
```

Design warnings

5/2/2020 • 15 minutes to read • Edit Online

Design warnings support adherence to the .NET Framework Design Guidelines.

In This Section

RULE	DESCRIPTION
CA1000: Do not declare static members on generic types	When a static member of a generic type is called, the type argument must be specified for the type. When a generic instance member that does not support inference is called, the type argument must be specified for the member. In these two cases, the syntax for specifying the type argument is different and easily confused.
CA1001: Types that own disposable fields should be disposable	A class declares and implements an instance field that is a System.IDisposable type and the class does not implement IDisposable. A class that declares an IDisposable field indirectly owns an unmanaged resource and should implement the IDisposable interface.
CA1002: Do not expose generic lists	System.Collections.Generic.List<(Of <(T>)>) is a generic collection that is designed for performance, not inheritance. Therefore, List does not contain any virtual members. The generic collections that are designed for inheritance should be exposed instead.
CA1003: Use generic event handler instances	A type contains a delegate that returns void, whose signature contains two parameters (the first an object and the second a type that is assignable to EventArgs), and the containing assembly targets .NET Framework 2.0.
CA1004: Generic methods should provide type parameter	Inference is how the type argument of a generic method is determined by the type of argument that is passed to the method, instead of by the explicit specification of the type argument. To enable inference, the parameter signature of a generic method must include a parameter that is of the same type as the type parameter for the method. In this case, the type argument does not have to be specified. When you use inference for all type parameters, the syntax for calling generic and nongeneric instance methods is identical; this simplifies the usability of generic methods.
CA1005: Avoid excessive parameters on generic types	The more type parameters a generic type contains, the more difficult it is to know and remember what each type parameter represents. It is usually obvious with one type parameter, as in List<T>, and in certain cases with two type parameters, as in Dictionary<TKey, TValue>. However, if more than two type parameters exist, the difficulty becomes too great for most users.

RULE	DESCRIPTION
CA1006: Do not nest generic types in member signatures	A nested type argument is a type argument that is also a generic type. To call a member whose signature contains a nested type argument, the user must instantiate one generic type and pass this type to the constructor of a second generic type. The required procedure and syntax are complex and should be avoided.
CA1007: Use generics where appropriate	An externally visible method contains a reference parameter of type System.Object. Use of a generic method enables all types, subject to constraints, to be passed to the method without first casting the type to the reference parameter type.
CA1008: Enums should have zero value	The default value of an uninitialized enumeration, just as other value types, is zero. A nonflags attributed enumeration should define a member by using the value of zero so that the default value is a valid value of the enumeration. If an enumeration that has the FlagsAttribute attribute applied defines a zero-valued member, its name should be "None" to indicate that no values have been set in the enumeration.
CA1009: Declare event handlers correctly	Event handler methods take two parameters. The first is of type System.Object and is named "sender". This is the object that raised the event. The second parameter is of type System.EventArgs and is named "e". This is the data that is associated with the event. Event handler methods should not return a value; in the C# programming language, this is indicated by the return type void.
CA1010: Collections should implement generic interface	To broaden the usability of a collection, implement one of the generic collection interfaces. Then the collection can be used to populate generic collection types.
CA1011: Consider passing base types as parameters	When a base type is specified as a parameter in a method declaration, any type that is derived from the base type can be passed as the corresponding argument to the method. If the additional functionality that is provided by the derived parameter type is not required, use of the base type enables wider use of the method.
CA1012: Abstract types should not have constructors	Constructors on abstract types can be called only by derived types. Because public constructors create instances of a type, and you cannot create instances of an abstract type, an abstract type that has a public constructor is incorrectly designed.
CA1013: Overload operator equals on overloading add and subtract	A public or protected type implements the addition or subtraction operators without implementing the equality operator.
CA1014: Mark assemblies with CLSCompliantAttribute	The Common Language Specification (CLS) defines naming restrictions, data types, and rules to which assemblies must conform if they will be used across programming languages. Good design dictates that all assemblies explicitly indicate CLS compliance by using CLSCompliantAttribute. If this attribute is not present on an assembly, the assembly is not compliant.

RULE	DESCRIPTION
CA1016: Mark assemblies with AssemblyVersionAttribute	.NET uses the version number to uniquely identify an assembly, and to bind to types in strongly named assemblies. The version number is used together with version and publisher policy. By default, applications run only with the assembly version with which they were built.
CA1017: Mark assemblies with ComVisibleAttribute	ComVisibleAttribute determines how COM clients access managed code. Good design dictates that assemblies explicitly indicate COM visibility. COM visibility can be set for the whole assembly and then overridden for individual types and type members. If this attribute is not present, the contents of the assembly are visible to COM clients.
CA1018: Mark attributes with AttributeUsageAttribute	When you define a custom attribute, mark it by using AttributeUsageAttribute to indicate where in the source code the custom attribute can be applied. The meaning and intended usage of an attribute will determine its valid locations in code.
CA1019: Define accessors for attribute arguments	Attributes can define mandatory arguments that must be specified when you apply the attribute to a target. These are also known as positional arguments because they are supplied to attribute constructors as positional parameters. For every mandatory argument, the attribute should also provide a corresponding read-only property so that the value of the argument can be retrieved at execution time. Attributes can also define optional arguments, which are also known as named arguments. These arguments are supplied to attribute constructors by name and should have a corresponding read/write property.
CA1020: Avoid namespaces with few types	Make sure that each of your namespaces has a logical organization, and that you have a valid reason to put types in a sparsely populated namespace.
CA1021: Avoid out parameters	Passing types by reference (using out or ref) requires experience with pointers, understanding how value types and reference types differ, and handling methods with multiple return values. Also, the difference between out and ref parameters is not widely understood.
CA1023: Indexers should not be multidimensional	Indexers (that is, indexed properties) should use a single index. Multidimensional indexers can significantly reduce the usability of the library.
CA1024: Use properties where appropriate	A public or protected method has a name that starts with "Get", takes no parameters, and returns a value that is not an array. The method might be a good candidate to become a property.
CA1025: Replace repetitive arguments with params array	Use a parameter array instead of repeated arguments when the exact number of arguments is unknown and when the variable arguments are the same type or can be passed as the same type.

RULE	DESCRIPTION
CA1026: Default parameters should not be used	Methods that use default parameters are allowed under the CLS; however, the CLS lets compilers ignore the values that are assigned to these parameters. To maintain the behavior that you want across programming languages, methods that use default parameters should be replaced by method overloads that provide the default parameters.
CA1027: Mark enums with FlagsAttribute	An enumeration is a value type that defines a set of related named constants. Apply FlagsAttribute to an enumeration when its named constants can be meaningfully combined.
CA1028: Enum storage should be Int32	An enumeration is a value type that defines a set of related named constants. By default, the System.Int32 data type is used to store the constant value. Even though you can change this underlying type, it is not required or recommended for most scenarios.
CA1030: Use events where appropriate	This rule detects methods that have names that ordinarily would be used for events. If a method is called in response to a clearly defined state change, the method should be invoked by an event handler. Objects that call the method should raise events instead of calling the method directly.
CA1031: Do not catch general exception types	General exceptions should not be caught. Catch a more-specific exception, or rethrow the general exception as the last statement in the catch block.
CA1032: Implement standard exception constructors	Failure to provide the full set of constructors can make it difficult to correctly handle exceptions.
CA1033: Interface methods should be callable by child types	An unsealed externally visible type provides an explicit method implementation of a public interface and does not provide an alternative externally visible method that has the same name.
CA1034: Nested types should not be visible	A nested type is a type that is declared in the scope of another type. Nested types are useful to encapsulate private implementation details of the containing type. Used for this purpose, nested types should not be externally visible.
CA1035: ICollection implementations have strongly typed members	This rule requires ICollection implementations to provide strongly typed members so that users are not required to cast arguments to the Object type when they use the functionality that is provided by the interface. This rule assumes that the type that implements ICollection does so to manage a collection of instances of a type that is stronger than Object.
CA1036: Override methods on comparable types	A public or protected type implements the System.IComparable interface. It does not override Object.Equals nor does it overload the language-specific operator for equality, inequality, less than, or greater than.

RULE	DESCRIPTION
CA1038: Enumerators should be strongly typed	This rule requires IEnumerator implementations to also provide a strongly typed version of the Current property so that users are not required to cast the return value to the strong type when they use the functionality that is provided by the interface.
CA1039: Lists are strongly typed	This rule requires IList implementations to provide strongly typed members so that users are not required to cast arguments to the System.Object type when they use the functionality that is provided by the interface.
CA1040: Avoid empty interfaces	Interfaces define members that provide a behavior or usage contract. The functionality that is described by the interface can be adopted by any type, regardless of where the type appears in the inheritance hierarchy. A type implements an interface by providing implementations for the members of the interface. An empty interface does not define any members; therefore, it does not define a contract that can be implemented.
CA1041: Provide ObsoleteAttribute message	A type or member is marked by using a System.ObsoleteAttribute attribute that does not have its ObsoleteAttribute.Message property specified. When a type or member that is marked by using ObsoleteAttribute is compiled, the Message property of the attribute is displayed, which gives the user information about the obsolete type or member.
CA1043: Use integral or string argument for indexers	Indexers (that is, indexed properties) should use integral or string types for the index. These types are typically used for indexing data structures and they increase the usability of the library. Use of the Object type should be restricted to those cases where the specific integral or string type cannot be specified at design time.
CA1044: Properties should not be write only	Although it is acceptable and often necessary to have a read-only property, the design guidelines prohibit the use of write-only properties. This is because letting a user set a value, and then preventing the user from viewing that value, does not provide any security. Also, without read access, the state of shared objects cannot be viewed, which limits their usefulness.
CA1045: Do not pass types by reference	Passing types by reference (using out or ref) requires experience with pointers, understanding how value types and reference types differ, and handling methods with multiple return values. Library architects who design for a general audience should not expect users to master working with out or ref parameters.
CA1046: Do not overload operator equals on reference types	For reference types, the default implementation of the equality operator is almost always correct. By default, two references are equal only if they point to the same object.
CA1047: Do not declare protected members in sealed types	Types declare protected members so that inheriting types can access or override the member. By definition, sealed types cannot be inherited, which means that protected methods on sealed types cannot be called.

RULE	DESCRIPTION
CA1048: Do not declare virtual members in sealed types	Types declare methods as virtual so that inheriting types can override the implementation of the virtual method. By definition, a sealed type cannot be inherited. This makes a virtual method on a sealed type meaningless.
CA1049: Types that own native resources should be disposable	Types that allocate unmanaged resources should implement IDisposable to enable callers to release those resources on demand and to shorten the lifetimes of the objects that hold the resources.
CA1050: Declare types in namespaces	Types are declared in namespaces to prevent name collisions and as a way to organize related types in an object hierarchy.
CA1051: Do not declare visible instance fields	The primary use of a field should be as an implementation detail. Fields should be private or internal and should be exposed by using properties.
CA1052: Static holder types should be sealed	A public or protected type contains only static members and is not declared by using the sealed (C#) or NotInheritable (Visual Basic) modifier. A type that is not meant to be inherited should be marked by using the sealed modifier to prevent its use as a base type.
CA1053: Static holder types should not have constructors	A public or nested public type declares only static members and has a public or protected default constructor. The constructor is unnecessary because calling static members does not require an instance of the type. The string overload should call the uniform resource identifier (URI) overload by using the string argument for safety and security.
CA1054: URI parameters should not be strings	If a method takes a string representation of a URI, a corresponding overload should be provided that takes an instance of the URI class, which provides these services in a safe and secure manner.
CA1055: URI return values should not be strings	This rule assumes that the method returns a URI. A string representation of a URI is prone to parsing and encoding errors, and can lead to security vulnerabilities. The System.Uri class provides these services in a safe and secure manner.
CA1056: URI properties should not be strings	This rule assumes that the property represents a URI. A string representation of a URI is prone to parsing and encoding errors, and can lead to security vulnerabilities. The System.Uri class provides these services in a safe and secure manner.
CA1057: String URI overloads call System.Uri overloads	A type declares method overloads that differ only by the replacement of a string parameter with a System.Uri parameter. The overload that takes the string parameter does not call the overload that takes the URI parameter.
CA1058: Types should not extend certain base types	An externally visible type extends certain base types. Use one of the alternatives.

RULE	DESCRIPTION
CA1059: Members should not expose certain concrete types	A concrete type is a type that has a complete implementation and therefore can be instantiated. To enable widespread use of the member, replace the concrete type by using the suggested interface.
CA1060: Move P/Invokes to NativeMethods class	Platform Invocation methods, such as those marked with the System.Runtime.InteropServices.DllImportAttribute or methods defined by using the Declare keyword in Visual Basic, access unmanaged code. These methods should be of the NativeMethods, SafeNativeMethods, or UnsafeNativeMethods class.
CA1061: Do not hide base class methods	A method in a base type is hidden by an identically named method in a derived type, when the parameter signature of the derived method differs only by types that are more weakly derived than the corresponding types in the parameter signature of the base method.
CA1062: Validate arguments of public methods	All reference arguments that are passed to externally visible methods should be checked against null.
CA1063: Implement IDisposable correctly	All IDisposable types should implement the Dispose pattern correctly.
CA1064: Exceptions should be public	An internal exception is visible only inside its own internal scope. After the exception falls outside the internal scope, only the base exception can be used to catch the exception. If the internal exception is inherited from System.Exception, System.SystemException, or System.ApplicationException, the external code will not have sufficient information to know what to do with the exception.
CA1065: Do not raise exceptions in unexpected locations	A method that is not expected to throw exceptions throws an exception.
CA1066: Implement IEquatable when overriding Equals	A value type overrides Equals method, but does not implement IEquatable<T>.
CA1067: Override Equals when implementing IEquatable	A type implements IEquatable<T>, but does not override Equals method.
CA1068: CancellationToken parameters must come last	A method has a CancellationToken parameter that is not the last parameter.
CA1069: Enums should not have duplicate values	An enumeration has multiple members which are explicitly assigned the same constant value.
CA1070: Do not declare event fields as virtual	A field-like event was declared as virtual.

RULE	DESCRIPTION
CA2210: Assemblies should have valid strong names	The strong name protects clients from unknowingly loading an assembly that has been tampered with. Assemblies without strong names should not be deployed outside very limited scenarios. If you share or distribute assemblies that are not correctly signed, the assembly can be tampered with, the common language runtime might not load the assembly, or the user might have to disable verification on his or her computer.

CA1000: Do not declare static members on generic types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1000
Category	Microsoft.Design
Breaking change	Breaking

Cause

A generic type contains a `static` (`Shared` in Visual Basic) member.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

When a `static` member of a generic type is called, the type argument must be specified for the type. When a generic instance member that does not support inference is called, the type argument must be specified for the member. The syntax for specifying the type argument in these two cases is different and easily confused, as the following calls demonstrate:

```
' Shared method in a generic type.
GenericType(Of Integer).SharedMethod()

' Generic instance method that does not support inference.
someObject.GenericMethod(Of Integer)()
```

```
// Static method in a generic type.
GenericType<int>.StaticMethod();

// Generic instance method that does not support inference.
someObject.GenericMethod<int>();
```

Generally, both of the prior declarations should be avoided so that the type argument does not have to be specified when the member is called. This results in a syntax for calling members in generics that is no different from the syntax for non-generics. For more information, see CA1004: Generic methods should provide type parameter.

How to fix violations

To fix a violation of this rule, remove the static member or change it to an instance member.

When to suppress warnings

Do not suppress a warning from this rule. Providing generics in a syntax that is easy to understand and use

reduces the time that is required to learn and increases the adoption rate of new libraries.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1000.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Related rules

- CA1005: Avoid excessive parameters on generic types
- CA1010: Collections should implement generic interface
- CA1002: Do not expose generic lists
- CA1006: Do not nest generic types in member signatures
- CA1004: Generic methods should provide type parameter
- CA1003: Use generic event handler instances
- CA1007: Use generics where appropriate

See also

- Generics

CA1001: Types that own disposable fields should be disposable

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1001
Category	Microsoft.Design
Breaking change	Non-breaking - If the type is not visible outside the assembly. Breaking - If the type is visible outside the assembly.

Cause

A class declares and implements an instance field that is a System.IDisposable type and the class does not implement IDisposable.

Rule description

A class implements the IDisposable interface to dispose of unmanaged resources that it owns. An instance field that is an IDisposable type indicates that the field owns an unmanaged resource. A class that declares an IDisposable field indirectly owns an unmanaged resource and should implement the IDisposable interface. If the class does not directly own any unmanaged resources, it should not implement a finalizer.

Starting with FxCop analyzers 3.x, this rule respects types implementing System.IAsyncDisposable as disposable types.

How to fix violations

To fix a violation of this rule, implement IDisposable and from the System.IDisposable.Dispose method call the Dispose method of the field.

When to suppress warnings

In general, do not suppress a warning from this rule. It is OK to suppress the warning when the dispose ownership of the field(s) is not held by the containing type.

Example

The following example shows a class that violates the rule and a class that satisfies the rule by implementing IDisposable. The class does not implement a finalizer because the class does not directly own any unmanaged resources.

```
Imports System
Imports System.IO

Namespace DesignLibrary

   ' This class violates the rule.
   Public Class NoDisposeMethod

      Dim newFile As FileStream

      Sub New()
         newFile = New FileStream("c:\temp.txt", FileMode.Open)
      End Sub

   End Class

   ' This class satisfies the rule.
   Public Class HasDisposeMethod
      Implements IDisposable

      Dim newFile As FileStream

      Sub New()
         newFile = New FileStream("c:\temp.txt", FileMode.Open)
      End Sub

      Overloads Protected Overridable Sub Dispose(disposing As Boolean)

         If disposing Then
            ' dispose managed resources
            newFile.Close()
         End If

         ' free native resources

      End Sub 'Dispose


      Overloads Public Sub Dispose() Implements IDisposable.Dispose

         Dispose(True)
         GC.SuppressFinalize(Me)

      End Sub 'Dispose

   End Class

End Namespace
```

```
using System;
using System.IO;

namespace DesignLibrary
{
   // This class violates the rule.
   public class NoDisposeMethod
   {
      FileStream newFile;

      public NoDisposeMethod()
      {
         newFile = new FileStream(@"c:\temp.txt", FileMode.Open);
      }
   }

   // This class satisfies the rule.
   public class HasDisposeMethod: IDisposable
   {
      FileStream newFile;

      public HasDisposeMethod()
      {
         newFile = new FileStream(@"c:\temp.txt", FileMode.Open);
      }

      protected virtual void Dispose(bool disposing)
      {
         if (disposing)
            {
               // dispose managed resources
               newFile.Close();
            }
          // free native resources
      }

      public void Dispose()
      {
         Dispose(true);
         GC.SuppressFinalize(this);
      }
   }
}
```

Related rules

CA2213: Disposable fields should be disposed

CA2216: Disposable types should declare finalizer

CA2215: Dispose methods should call base class dispose

CA1049: Types that own native resources should be disposable

CA1002: Do not expose generic lists

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1002
Category	Microsoft.Design
Breaking change	Breaking

Cause

A type contains an externally visible member that is a System.Collections.Generic.List<T> type, returns a List<T> type, or whose signature includes a List<T> parameter.

Rule description

System.Collections.Generic.List<T> is a generic collection that's designed for performance and not inheritance. List<T> does not contain virtual members that make it easier to change the behavior of an inherited class. The following generic collections are designed for inheritance and should be exposed instead of List<T>.

- System.Collections.ObjectModel.Collection<T>
- System.Collections.ObjectModel.ReadOnlyCollection<T>
- System.Collections.ObjectModel.KeyedCollection<TKey,TItem>
- System.Collections.Generic.IList<T>
- System.Collections.Generic.ICollection<T>

How to fix violations

To fix a violation of this rule, change the System.Collections.Generic.List<T> type to one of the generic collections that's designed for inheritance.

When to suppress warnings

Do not suppress a warning from this rule unless the assembly that raises this warning is not meant to be a reusable library. For example, it would be safe to suppress this warning in a performance-tuned application where a performance benefit was gained from the use of generic lists.

Related rules

CA1005: Avoid excessive parameters on generic types

CA1010: Collections should implement generic interface

CA1000: Do not declare static members on generic types

CA1006: Do not nest generic types in member signatures

CA1004: Generic methods should provide type parameter

CA1003: Use generic event handler instances

CA1007: Use generics where appropriate

See also

Generics

CA1003: Use generic event handler instances

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1003
Category	Microsoft.Design
Breaking change	Breaking

Cause

A type contains a delegate that returns void and whose signature contains two parameters (the first an object and the second a type that is assignable to EventArgs), and the containing assembly targets .NET.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

Before .NET Framework 2.0, in order to pass custom information to the event handler, a new delegate had to be declared that specified a class that was derived from the System.EventArgs class. In .NET Framework 2.0 and later versions, the generic System.EventHandler<TEventArgs> delegate allows any class that's derived from EventArgs to be used together with the event handler.

How to fix violations

To fix a violation of this rule, remove the delegate and replace its use by using the System.EventHandler<TEventArgs> delegate.

If the delegate is autogenerated by the Visual Basic compiler, change the syntax of the event declaration to use the System.EventHandler<TEventArgs> delegate.

When to suppress warnings

Do not suppress a warning from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1003.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows a delegate that violates the rule. In the Visual Basic example, comments describe how to modify the example to satisfy the rule. For the C# example, an example follows that shows the modified code.

```
Imports System

Namespace DesignLibrary

   Public Class CustomEventArgs
      Inherits EventArgs

      Public info As String = "data"

   End Class

   Public Class ClassThatRaisesEvent

      ' This statement creates a new delegate, which violates the rule.
      Event SomeEvent(sender As Object, e As CustomEventArgs)

      ' To satisfy the rule, comment out the previous line
      ' and uncomment the following line.
      'Event SomeEvent As EventHandler(Of CustomEventArgs)

      Protected Overridable Sub OnSomeEvent(e As CustomEventArgs)
            RaiseEvent SomeEvent(Me, e)
      End Sub

      Sub SimulateEvent()
         OnSomeEvent(New CustomEventArgs())
      End Sub

   End Class

   Public Class ClassThatHandlesEvent

      Sub New(eventRaiser As ClassThatRaisesEvent)
         AddHandler eventRaiser.SomeEvent, AddressOf HandleEvent
      End Sub

      Private Sub HandleEvent(sender As Object, e As CustomEventArgs)
         Console.WriteLine("Event handled: {0}", e.info)
      End Sub

   End Class

   Class Test

      Shared Sub Main()

         Dim eventRaiser As New ClassThatRaisesEvent()
         Dim eventHandler As New ClassThatHandlesEvent(eventRaiser)

         eventRaiser.SimulateEvent()

      End Sub

   End Class

End Namespace
```

```
using System;

namespace DesignLibrary
{
   // This delegate violates the rule.
   public delegate void CustomEventHandler(
      object sender, CustomEventArgs e);

   public class CustomEventArgs : EventArgs
   {
      public string info = "data";
   }

   public class ClassThatRaisesEvent
   {
      public event CustomEventHandler SomeEvent;

      protected virtual void OnSomeEvent(CustomEventArgs e)
      {
         if(SomeEvent != null)
         {
            SomeEvent(this, e);
         }
      }

      public void SimulateEvent()
      {
         OnSomeEvent(new CustomEventArgs());
      }
   }

   public class ClassThatHandlesEvent
   {
      public ClassThatHandlesEvent(ClassThatRaisesEvent eventRaiser)
      {
         eventRaiser.SomeEvent +=
            new CustomEventHandler(HandleEvent);
      }

      private void HandleEvent(object sender, CustomEventArgs e)
      {
         Console.WriteLine("Event handled: {0}", e.info);
      }
   }

   class Test
   {
      static void Main()
      {
         ClassThatRaisesEvent eventRaiser = new ClassThatRaisesEvent();
         ClassThatHandlesEvent eventHandler =
            new ClassThatHandlesEvent(eventRaiser);

         eventRaiser.SimulateEvent();
      }
   }
}
```

The following code snippet removes the delegate declaration from the previous example, which satisfies the rule. It replaces its use in the `ClassThatRaisesEvent` and `ClassThatHandlesEvent` methods by using the System.EventHandler<TEventArgs> delegate.

```
using System;

namespace DesignLibrary
{
   public class CustomEventArgs : EventArgs
   {
      public string info = "data";
   }

   public class ClassThatRaisesEvent
   {
      public event EventHandler<CustomEventArgs> SomeEvent;

      protected virtual void OnSomeEvent(CustomEventArgs e)
      {
         if(SomeEvent != null)
         {
            SomeEvent(this, e);
         }
      }

      public void SimulateEvent()
      {
         OnSomeEvent(new CustomEventArgs());
      }
   }

   public class ClassThatHandlesEvent
   {
      public ClassThatHandlesEvent(ClassThatRaisesEvent eventRaiser)
      {
         eventRaiser.SomeEvent +=
            new EventHandler<CustomEventArgs>(HandleEvent);
      }

      private void HandleEvent(object sender, CustomEventArgs e)
      {
         Console.WriteLine("Event handled: {0}", e.info);
      }
   }

   class Test
   {
      static void Main()
      {
         ClassThatRaisesEvent eventRaiser = new ClassThatRaisesEvent();
         ClassThatHandlesEvent eventHandler =
            new ClassThatHandlesEvent(eventRaiser);

         eventRaiser.SimulateEvent();
      }
   }
}
```

Related rules

- CA1005: Avoid excessive parameters on generic types
- CA1010: Collections should implement generic interface
- CA1000: Do not declare static members on generic types
- CA1002: Do not expose generic lists
- CA1006: Do not nest generic types in member signatures
- CA1004: Generic methods should provide type parameter

- CA1007: Use generics where appropriate

See also

- Generics

CA1004: Generic methods should provide type parameter

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1004
Category	Microsoft.Design
Breaking change	Breaking

Cause

The parameter signature of an externally visible generic method does not contain types that correspond to all the type parameters of the method.

Rule description

Inference is how the type argument of a generic method is determined by the type of argument that is passed to the method, instead of by the explicit specification of the type argument. To enable inference, the parameter signature of a generic method must include a parameter that is of the same type as the type parameter for the method. In this case, the type argument does not have to be specified. When you use inference for all type parameters, the syntax for calling generic and nongeneric instance methods is identical. This simplifies the usability of generic methods.

How to fix violations

To fix a violation of this rule, change the design so that the parameter signature contains the same type for each type parameter of the method.

When to suppress warnings

Do not suppress a warning from this rule. Providing generics in a syntax that is easy to understand and use reduces the time that is required to learn and increases the adoption rate of new libraries.

Example

The following example shows the syntax for calling two generic methods. The type argument for `InferredTypeArgument` is inferred, and the type argument for `NotInferredTypeArgument` must be explicitly specified.

```
Imports System

Namespace DesignLibrary

   Public Class Inference

      ' This method violates the rule.
      Sub NotInferredTypeArgument(Of T)()

         Console.WriteLine(GetType(T))

      End Sub

      ' This method satisfies the rule.
      Sub InferredTypeArgument(Of T)(sameAsTypeParameter As T)

         Console.WriteLine(sameAsTypeParameter)

      End Sub

   End Class

   Class Test

      Shared Sub Main()

         Dim infer As New Inference()
         infer.NotInferredTypeArgument(Of Integer)()
         infer.InferredTypeArgument(3)

      End Sub

   End Class

End Namespace
```

```
using System;

namespace DesignLibrary
{
   public class Inference
   {
      // This method violates the rule.
      public void NotInferredTypeArgument<T>()
      {
         Console.WriteLine(typeof(T));
      }

      // This method satisfies the rule.
      public void InferredTypeArgument<T>(T sameAsTypeParameter)
      {
         Console.WriteLine(sameAsTypeParameter);
      }
   }

   class Test
   {
      static void Main()
      {
         Inference infer = new Inference();
         infer.NotInferredTypeArgument<int>();
         infer.InferredTypeArgument(3);
      }
   }
}
```

Related rules

CA1005: Avoid excessive parameters on generic types

CA1010: Collections should implement generic interface

CA1000: Do not declare static members on generic types

CA1002: Do not expose generic lists

CA1006: Do not nest generic types in member signatures

CA1003: Use generic event handler instances

CA1007: Use generics where appropriate

See also

Generics

CA1005: Avoid excessive parameters on generic types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1005
Category	Microsoft.Design
Breaking change	Breaking

Cause

An externally visible generic type has more than two type parameters.

Rule description

The more type parameters a generic type contains, the more difficult it is to know and remember what each type parameter represents. It is usually obvious with one type parameter, as in `List<T>`, and in certain cases with two type parameters, as in `Dictionary<TKey, TValue>`. If more than two type parameters exist, the difficulty becomes too great for most users (for example, `TooManyTypeParameters<T, K, V>` in C# or `TooManyTypeParameters(Of T, K, V)` in Visual Basic).

How to fix violations

To fix a violation of this rule, change the design to use no more than two type parameters.

When to suppress warnings

Do not suppress a warning from this rule unless the design absolutely requires more than two type parameters. Providing generics in a syntax that is easy to understand and use reduces the time that is required to learn and increases the adoption rate of new libraries.

Configurability

Starting with version 3.0.0 of the analyzer package, If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1005.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Related rules

CA1010: Collections should implement generic interface

CA1000: Do not declare static members on generic types

CA1002: Do not expose generic lists

CA1006: Do not nest generic types in member signatures

CA1004: Generic methods should provide type parameter

CA1003: Use generic event handler instances

CA1007: Use generics where appropriate

See also

Generics

CA1006: Do not nest generic types in member signatures

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1006
Category	Microsoft.Design
Breaking change	Breaking

Cause

An externally visible member has a signature that contains a nested type argument.

Rule description

A nested type argument is a type argument that is also a generic type. To call a member whose signature contains a nested type argument, the user must instantiate one generic type and pass this type to the constructor of a second generic type. The required procedure and syntax are complex and should be avoided.

How to fix violations

To fix a violation of this rule, change the design to remove the nested type argument.

When to suppress warnings

Do not suppress a warning from this rule. Providing generics in a syntax that is easy to understand and use reduces the time that is required to learn and increases the adoption rate of new libraries.

Example

The following example shows a method that violates the rule and the syntax that is required to call the violating method.

```
Imports System
Imports System.Collections.Generic

Namespace DesignLibrary

   Public Class IntegerCollections

      Sub NonNestedCollection(collection As ICollection(Of Integer))

         For Each I As Integer In DirectCast( _
            collection, IEnumerable(Of Integer))

            Console.WriteLine(I)

         Next
```

```
      End Sub

      ' This method violates the rule.
      Sub NestedCollection( _
         outerCollection As ICollection(Of ICollection(Of Integer)))

         For Each innerCollection As ICollection(Of Integer) In _
            DirectCast(outerCollection, _
                       IEnumerable(Of ICollection(Of Integer)))

            For Each I As Integer In _
               DirectCast(innerCollection, IEnumerable(Of Integer))

               Console.WriteLine(I)

            Next

         Next

      End Sub

   End Class

   Class Test

      Shared Sub Main()

         Dim collections As New IntegerCollections()

         Dim integerListA As New List(Of Integer)()
         integerListA.Add(1)
         integerListA.Add(2)
         integerListA.Add(3)

         collections.NonNestedCollection(integerListA)

         Dim integerListB As New List(Of Integer)()
         integerListB.Add(4)
         integerListB.Add(5)
         integerListB.Add(6)

         Dim integerListC As New List(Of Integer)()
         integerListC.Add(7)
         integerListC.Add(8)
         integerListC.Add(9)

         Dim nestedIntegerLists As New List(Of ICollection(Of Integer))()
         nestedIntegerLists.Add(integerListA)
         nestedIntegerLists.Add(integerListB)
         nestedIntegerLists.Add(integerListC)

         collections.NestedCollection(nestedIntegerLists)

      End Sub

   End Class

End Namespace
```

```
using System;
using System.Collections.Generic;

namespace DesignLibrary
{
   public class IntegerCollections
   {
      public void NotNestedCollection(ICollection<int> collection)
      {
         foreach(int i in collection)
         {
            Console.WriteLine(i);
         }
      }

      // This method violates the rule.
      public void NestedCollection(
         ICollection<ICollection<int>> outerCollection)
      {
         foreach(ICollection<int> innerCollection in outerCollection)
         {
            foreach(int i in innerCollection)
            {
               Console.WriteLine(i);
            }
         }
      }
   }

   class Test
   {
      static void Main()
      {
         IntegerCollections collections = new IntegerCollections();

         List<int> integerListA = new List<int>();
         integerListA.Add(1);
         integerListA.Add(2);
         integerListA.Add(3);

         collections.NotNestedCollection(integerListA);

         List<int> integerListB = new List<int>();
         integerListB.Add(4);
         integerListB.Add(5);
         integerListB.Add(6);

         List<int> integerListC = new List<int>();
         integerListC.Add(7);
         integerListC.Add(8);
         integerListC.Add(9);

         List<ICollection<int>> nestedIntegerLists =
            new List<ICollection<int>>();
         nestedIntegerLists.Add(integerListA);
         nestedIntegerLists.Add(integerListB);
         nestedIntegerLists.Add(integerListC);

         collections.NestedCollection(nestedIntegerLists);
      }
   }
}
```

Related rules

CA1005: Avoid excessive parameters on generic types

CA1010: Collections should implement generic interface

CA1000: Do not declare static members on generic types

CA1002: Do not expose generic lists

CA1004: Generic methods should provide type parameter

CA1003: Use generic event handler instances

CA1007: Use generics where appropriate

See also

Generics

CA1007: Use generics where appropriate

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1007
Category	Microsoft.Design
Breaking change	Breaking

Cause

An externally visible method contains a reference parameter of type System.Object, and the containing assembly targets .NET Framework 2.0.

Rule description

A reference parameter is a parameter that is modified by using the `ref` (`ByRef` in Visual Basic) keyword. The argument type that is supplied for a reference parameter must exactly match the reference parameter type. To use a type that is derived from the reference parameter type, the type must first be cast and assigned to a variable of the reference parameter type. Use of a generic method allows all types, subject to constraints, to be passed to the method without first casting the type to the reference parameter type.

How to fix violations

To fix a violation of this rule, make the method generic and replace the Object parameter by using a type parameter.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a general-purpose swap routine that is implemented as both nongeneric and generic methods. Note how efficiently the strings are swapped by using the generic method compared to the nongeneric method.

```
Imports System

Namespace DesignLibrary

   Public NotInheritable Class ReferenceParameters

      Private Sub New()
      End Sub

      ' This method violates the rule.
      Public Shared Sub Swap( _
         ByRef object1 As Object, ByRef object2 As Object)

         Dim temp As Object = object1
         object1 = object2
         object2 = temp

      End Sub

      ' This method satifies the rule.
      Public Shared Sub GenericSwap(Of T)( _
         ByRef reference1 As T, ByRef reference2 As T)

         Dim temp As T = reference1
         reference1 = reference2
         reference2 = temp

      End Sub

   End Class

   Class Test

      Shared Sub Main()

         Dim string1 As String = "Swap"
         Dim string2 As String = "It"

         Dim object1 As Object = DirectCast(string1, Object)
         Dim object2 As Object = DirectCast(string2, Object)
         ReferenceParameters.Swap(object1, object2)
         string1 = DirectCast(object1, String)
         string2 = DirectCast(object2, String)
         Console.WriteLine("{0} {1}", string1, string2)

         ReferenceParameters.GenericSwap(string1, string2)
         Console.WriteLine("{0} {1}", string1, string2)

      End Sub

   End Class

End Namespace
```

```
using System;

namespace DesignLibrary
{
   public sealed class ReferenceParameters
   {
      private ReferenceParameters(){}

      // This method violates the rule.
      public static void Swap(ref object object1, ref object object2)
      {
         object temp = object1;
         object1 = object2;
         object2 = temp;
      }

      // This method satifies the rule.
      public static void GenericSwap<T>(ref T reference1, ref T reference2)
      {
         T temp = reference1;
         reference1 = reference2;
         reference2 = temp;
      }
   }

   class Test
   {
      static void Main()
      {
         string string1 = "Swap";
         string string2 = "It";

         object object1 = (object)string1;
         object object2 = (object)string2;
         ReferenceParameters.Swap(ref object1, ref object2);
         string1 = (string)object1;
         string2 = (string)object2;
         Console.WriteLine("{0} {1}", string1, string2);

         ReferenceParameters.GenericSwap(ref string1, ref string2);
         Console.WriteLine("{0} {1}", string1, string2);
      }
   }
}
```

Related rules

CA1005: Avoid excessive parameters on generic types

CA1010: Collections should implement generic interface

CA1000: Do not declare static members on generic types

CA1002: Do not expose generic lists

CA1006: Do not nest generic types in member signatures

CA1004: Generic methods should provide type parameter

CA1003: Use generic event handler instances

See also

Generics

CA1008: Enums should have zero value

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1008
Category	Microsoft.Design
Breaking change	Non-breaking - When you are prompted to add a **None** value to a non-flag enumeration. Breaking - When you are prompted to rename or remove any enumeration values.

Cause

An enumeration without an applied System.FlagsAttribute does not define a member that has a value of zero. Or, an enumeration that has an applied FlagsAttribute defines a member that has a value of zero but its name is not 'None'. Or, the enumeration defines multiple, zero-valued members.

By default, this rule only looks at externally visible enumerations, but this is configurable.

Rule description

The default value of an uninitialized enumeration, just like other value types, is zero. A non-flags-attributed enumeration should define a member that has the value of zero so that the default value is a valid value of the enumeration. If appropriate, name the member 'None'. Otherwise, assign zero to the most frequently used member. By default, if the value of the first enumeration member is not set in the declaration, its value is zero.

If an enumeration that has the FlagsAttribute applied defines a zero-valued member, its name should be 'None' to indicate that no values have been set in the enumeration. Using a zero-valued member for any other purpose is contrary to the use of the FlagsAttribute in that the AND and OR bitwise operators are useless with the member. This implies that only one member should be assigned the value zero. If multiple members that have the value zero occur in a flags-attributed enumeration, `Enum.ToString()` returns incorrect results for members that are not zero.

How to fix violations

To fix a violation of this rule for non-flags-attributed enumerations, define a member that has the value of zero; this is a non-breaking change. For flags-attributed enumerations that define a zero-valued member, name this member 'None' and delete any other members that have a value of zero; this is a breaking change.

When to suppress warnings

Do not suppress a warning from this rule except for flags-attributed enumerations that have previously shipped.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1008.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows two enumerations that satisfy the rule and an enumeration, `BadTraceOptions`, that violates the rule.

```
using namespace System;

namespace DesignLibrary
{
   public enum class TraceLevel
   {
      Off     = 0,
      Error   = 1,
      Warning = 2,
      Info    = 3,
      Verbose = 4
   };

   [Flags]
   public enum class TraceOptions
   {
      None         =    0,
      CallStack    = 0x01,
      LogicalStack = 0x02,
      DateTime     = 0x04,
      Timestamp    = 0x08
   };

   [Flags]
   public enum class BadTraceOptions
   {
      CallStack    =    0,
      LogicalStack = 0x01,
      DateTime     = 0x02,
      Timestamp    = 0x04
   };
}

using namespace DesignLibrary;

void main()
{
   // Set the flags.
   BadTraceOptions badOptions = safe_cast<BadTraceOptions>
      (BadTraceOptions::LogicalStack | BadTraceOptions::Timestamp);

   // Check whether CallStack is set.
   if((badOptions & BadTraceOptions::CallStack) ==
         BadTraceOptions::CallStack)
   {
      // This 'if' statement is always true.
   }
}
```

```
using System;

namespace DesignLibrary
{
   public enum TraceLevel
   {
      Off     = 0,
      Error   = 1,
      Warning = 2,
      Info    = 3,
      Verbose = 4
   }

   [Flags]
   public enum TraceOptions
   {
      None         =    0,
      CallStack    = 0x01,
      LogicalStack = 0x02,
      DateTime     = 0x04,
      Timestamp    = 0x08,
   }

   [Flags]
   public enum BadTraceOptions
   {
      CallStack    =    0,
      LogicalStack = 0x01,
      DateTime     = 0x02,
      Timestamp    = 0x04,
   }

   class UseBadTraceOptions
   {
      static void Main()
      {
         // Set the flags.
         BadTraceOptions badOptions =
            BadTraceOptions.LogicalStack | BadTraceOptions.Timestamp;

         // Check whether CallStack is set.
         if((badOptions & BadTraceOptions.CallStack) ==
             BadTraceOptions.CallStack)
         {
            // This 'if' statement is always true.
         }
      }
   }
}
```

```
Imports System

Namespace DesignLibrary

   Public Enum TraceLevel
      Off     = 0
      AnError = 1
      Warning = 2
      Info    = 3
      Verbose = 4
   End Enum

   <Flags> _
   Public Enum TraceOptions
      None         =    0
      CallStack    = &H01
      LogicalStack = &H02
      DateTime     = &H04
      Timestamp    = &H08
   End Enum

   <Flags> _
   Public Enum BadTraceOptions
      CallStack    =    0
      LogicalStack = &H01
      DateTime     = &H02
      Timestamp    = &H04
   End Enum

   Class UseBadTraceOptions

      Shared Sub Main()

         ' Set the flags.
         Dim badOptions As BadTraceOptions = _
            BadTraceOptions.LogicalStack Or BadTraceOptions.Timestamp

         ' Check whether CallStack is set.
         If((badOptions And BadTraceOptions.CallStack) = _
             BadTraceOptions.CallStack)
            ' This 'If' statement is always true.
         End If

      End Sub

   End Class

End Namespace
```

Related rules

- CA2217: Do not mark enums with FlagsAttribute
- CA1700: Do not name enum values 'Reserved'
- CA1712: Do not prefix enum values with type name
- CA1028: Enum storage should be Int32
- CA1027: Mark enums with FlagsAttribute

See also

- System.Enum

CA1009: Declare event handlers correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1009
Category	Microsoft.Design
Breaking change	Breaking

Cause

A delegate that handles a public or protected event does not have the correct signature, return type, or parameter names.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Event handler methods take two parameters. The first is of type System.Object and is named 'sender'. This is the object that raised the event. The second parameter is of type System.EventArgs and is named 'e'. This is the data that is associated with the event. For example, if the event is raised whenever a file is opened, the event data typically contains the name of the file.

Event handler methods should not return a value. In the C# programming language, this is indicated by the return type `void`. An event handler can invoke multiple methods in multiple objects. If the methods were allowed to return a value, multiple return values would occur for each event, and only the value of the last method that was invoked would be available.

How to fix violations

To fix a violation of this rule, correct the signature, return type, or parameter names of the delegate. For details, see the following example.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a delegate that is suited to handling events. The methods that can be invoked by this event handler comply with the signature that is specified in the Design Guidelines. `AlarmEventHandler` is the type name of the delegate. `AlarmEventArgs` derives from the base class for event data, EventArgs, and holds alarm event data.

```
using namespace System;

namespace DesignLibrary
{
   public ref class AlarmEventArgs : public EventArgs {};
   public delegate void AlarmEventHandler(
      Object^ sender, AlarmEventArgs^ e);
}
```

```
using System;

namespace DesignLibrary
{
   public class AlarmEventArgs : EventArgs {}
   public delegate void AlarmEventHandler(object sender, AlarmEventArgs e);
}
```

```
Imports System

Namespace DesignLibrary

   Public Delegate Sub AlarmEventHandler(sender As Object, e As AlarmEventArgs)

   Public Class AlarmEventArgs
      Inherits EventArgs
   End Class

End Namespace
```

Related rules

CA2109: Review visible event handlers

See also

- System.EventArgs
- System.Object
- Handling and raising events

CA1010: Collections should implement generic interface

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1010
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A type implements the System.Collections.IEnumerable interface but does not implement the System.Collections.Generic.IEnumerable<T> interface, and the containing assembly targets .NET. This rule ignores types that implement System.Collections.IDictionary.

By default, this rule only looks at externally visible types, but this is configurable. You can also configure additional interfaces to require that a generic interface be implemented.

Rule description

To broaden the usability of a collection, implement one of the generic collection interfaces. Then the collection can be used to populate generic collection types such as the following:

- System.Collections.Generic.List<T>
- System.Collections.Generic.Queue<T>
- System.Collections.Generic.Stack<T>

How to fix violations

To fix a violation of this rule, implement one of the following generic collection interfaces:

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.ICollection<T>
- System.Collections.Generic.IList<T>

When to suppress warnings

It is safe to suppress a warning from this rule; however, the use of the collection will be more limited.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1010.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Additional required generic interfaces

You can configure the list of interface names (separated by '|') with their required generic fully qualified interface (separated by '->').

Allowed interface formats:

- Interface name only (includes all interfaces with the name, regardless of the containing type or namespace).
- Fully qualified names in the symbol's documentation ID format: https://github.com/dotnet/csharplang/blob/master/spec/documentation-comments.md#id-string-format with an optional "T:" prefix.

Examples:

OPTION VALUE	SUMMARY

`dotnet_code_quality.CA1010.additional_required_generic_interfaces = ISomething->System.Collections.Generic.IEnumerable`
``1
`| All types implementing 'ISomething' regardless of its namespace are expected to also implement 'System.Collections.Generic.IEnumerable`
1'.	

`dotnet_code_quality.CA1010.additional_required_generic_interfaces = T:System.Collections.IDictionary->T:System.Collections.Generic.IDictionary`
``2
`| All types implementing 'System.Collections.Generic.IDictionary' are expected to also implement 'System.Collections.Generic.IDictionary`
2'.	

Example violation

The following example shows a class (reference type) that derives from the non-generic `CollectionBase` class, which violates this rule.

```
using System;
using System.Collections;

namespace Samples
{
    public class Book
    {
        public Book()
        {
        }
    }

    public class BookCollection : CollectionBase
    {
        public BookCollection()
        {
        }

        public void Add(Book value)
        {
            InnerList.Add(value);
        }

        public void Remove(Book value)
        {
            InnerList.Remove(value);
        }

        public void Insert(int index, Book value)
        {
            InnerList.Insert(index, value);
        }

        public Book this[int index]
        {
            get { return (Book)InnerList[index]; }
            set { InnerList[index] = value; }
        }

        public bool Contains(Book value)
        {
            return InnerList.Contains(value);
        }

        public int IndexOf(Book value)
        {
            return InnerList.IndexOf(value);
        }

        public void CopyTo(Book[] array, int arrayIndex)
        {
            InnerList.CopyTo(array, arrayIndex);
        }
    }
}
```

To fix a violation of this rule, do one of the following:

- Implement the generic interfaces.
- Change the base class to a type that already implements both the generic and non-generic interfaces, such as the `Collection<T>` class.

Fix by base class change

The following example fixes the violation by changing the base class of the collection from the non-generic

`CollectionBase` class to the generic `Collection<T>` (`Collection(Of T)` in Visual Basic) class.

```
using System;
using System.Collections.ObjectModel;

namespace Samples
{
    public class Book
    {
        public Book()
        {
        }
    }

    public class BookCollection : Collection<Book>
    {
        public BookCollection()
        {
        }
    }
}
```

Changing the base class of an already released class is considered a breaking change to existing consumers.

Fix by interface implementation

The following example fixes the violation by implementing these generic interfaces: `IEnumerable<T>`, `ICollection<T>`, and `IList<T>` (`IEnumerable(Of T)`, `ICollection(Of T)`, and `IList(Of T)` in Visual Basic).

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace Samples
{
    public class Book
    {
        public Book()
        {
        }
    }

    public class BookCollection : CollectionBase, IList<Book>
    {
        public BookCollection()
        {
        }

        int IList<Book>.IndexOf(Book item)
        {
            return this.List.IndexOf(item);
        }

        void IList<Book>.Insert(int location, Book item)
        {
        }

        Book IList<Book>.this[int index]
        {
            get { return (Book) this.List[index]; }
            set { }
        }

        void ICollection<Book>.Add(Book item)
```

```
        {
        }

        bool ICollection<Book>.Contains(Book item)
        {
            return true;
        }

        void ICollection<Book>.CopyTo(Book[] array, int arrayIndex)
        {
        }

        bool ICollection<Book>.IsReadOnly
        {
            get { return false; }
        }

        bool ICollection<Book>.Remove(Book item)
        {
            if (InnerList.Contains(item))
            {
                InnerList.Remove(item);
                return true;
            }
            return false;
        }

        IEnumerator<Book> IEnumerable<Book>.GetEnumerator()
        {
            return new BookCollectionEnumerator(InnerList.GetEnumerator());
        }

        private class BookCollectionEnumerator : IEnumerator<Book>
        {
            private IEnumerator _Enumerator;

            public BookCollectionEnumerator(IEnumerator enumerator)
            {
                _Enumerator = enumerator;
            }

            public Book Current
            {
                get { return (Book)_Enumerator.Current; }
            }

            object IEnumerator.Current
            {
                get { return _Enumerator.Current; }
            }

            public bool MoveNext()
            {
                return _Enumerator.MoveNext();
            }

            public void Reset()
            {
                _Enumerator.Reset();
            }

            public void Dispose()
            {
            }
        }
    }
}
```

Related rules

- CA1005: Avoid excessive parameters on generic types
- CA1000: Do not declare static members on generic types
- CA1002: Do not expose generic lists
- CA1006: Do not nest generic types in member signatures
- CA1004: Generic methods should provide type parameter
- CA1003: Use generic event handler instances
- CA1007: Use generics where appropriate

See also

- Generics

CA1011: Consider passing base types as parameters

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1011
Category	Microsoft.Design
Breaking change	Breaking

Cause

A method declaration includes a formal parameter that is a derived type, and the method calls only members of the base type of the parameter.

Rule description

When a base type is specified as a parameter in a method declaration, any type that is derived from the base type can be passed as the corresponding argument to the method. When the argument is used inside the method body, the specific method that is executed depends on the type of the argument. If the additional functionality that is provided by the derived type is not required, use of the base type allows wider use of the method.

How to fix violations

To fix a violation of this rule, change the type of the parameter to its base type.

When to suppress warnings

It is safe to suppress a warning from this rule

- if the method requires the specific functionality that is provided by the derived type

  - or -

- to enforce that only the derived type, or a more derived type, is passed to the method.

In these cases, the code will be more robust because of the strong type checking that is provided by the compiler and runtime.

Example

The following example shows a method, `ManipulateFileStream`, that can be used only with a FileStream object, which violates this rule. A second method, `ManipulateAnyStream`, satisfies the rule by replacing the FileStream parameter by using a Stream.

```
using System;
using System.IO;

namespace DesignLibrary
{
    public class StreamUser
    {
        int anInteger;

        public void ManipulateFileStream(FileStream stream)
        {
            while ((anInteger = stream.ReadByte()) != -1)
            {
                // Do something.
            }
        }

        public void ManipulateAnyStream(Stream anyStream)
        {
            while ((anInteger = anyStream.ReadByte()) != -1)
            {
                // Do something.
            }
        }
    }

    class TestStreams
    {
        static void Main()
        {
            StreamUser someStreamUser = new StreamUser();
            MemoryStream testMemoryStream = new MemoryStream(new byte[] { });
            using (FileStream testFileStream =
                     new FileStream("test.dat", FileMode.OpenOrCreate))
            {
                // Cannot be used with testMemoryStream.
                someStreamUser.ManipulateFileStream(testFileStream);

                someStreamUser.ManipulateAnyStream(testFileStream);
                someStreamUser.ManipulateAnyStream(testMemoryStream);
            }
        }
    }
}
```

```
using namespace System;
using namespace System::IO;

namespace DesignLibrary
{
   public ref class StreamUser
   {
      int anInteger;

   public:
      void ManipulateFileStream(FileStream^ stream)
      {
         while((anInteger = stream->ReadByte()) != -1)
         {
            // Do something.
         }
      }

      void ManipulateAnyStream(Stream^ anyStream)
      {
         while((anInteger = anyStream->ReadByte()) != -1)
         {
            // Do something.
         }
      }
   };
}

using namespace DesignLibrary;

static void main()
{
   StreamUser^ someStreamUser = gcnew StreamUser();
   FileStream^ testFileStream =
      gcnew FileStream("test.dat", FileMode::OpenOrCreate);
   MemoryStream^ testMemoryStream =
      gcnew MemoryStream(gcnew array<Byte>{});

   // Cannot be used with testMemoryStream.
   someStreamUser->ManipulateFileStream(testFileStream);

   someStreamUser->ManipulateAnyStream(testFileStream);
   someStreamUser->ManipulateAnyStream(testMemoryStream);

   testFileStream->Close();
}
```

```
Imports System
Imports System.IO

Namespace DesignLibrary

    Public Class StreamUser

        Sub ManipulateFileStream(ByVal stream As IO.FileStream)
            If stream Is Nothing Then Throw New ArgumentNullException("stream")

            Dim anInteger As Integer = stream.ReadByte()
            While (anInteger <> -1)
                ' Do something.
                anInteger = stream.ReadByte()
            End While
        End Sub

        Sub ManipulateAnyStream(ByVal anyStream As IO.Stream)
            If anyStream Is Nothing Then Throw New ArgumentNullException("anyStream")

            Dim anInteger As Integer = anyStream.ReadByte()
            While (anInteger <> -1)
                ' Do something.
                anInteger = anyStream.ReadByte()
            End While
        End Sub
    End Class


   Public Class TestStreams

      Shared Sub Main()
            Dim someStreamUser As New StreamUser()
            Dim testFileStream As New FileStream( _
               "test.dat", FileMode.OpenOrCreate)
            Dim testMemoryStream As New MemoryStream(New Byte() {})

            ' Cannot be used with testMemoryStream.
            someStreamUser.ManipulateFileStream(testFileStream)
            someStreamUser.ManipulateAnyStream(testFileStream)
            someStreamUser.ManipulateAnyStream(testMemoryStream)
            testFileStream.Close()
        End Sub
   End Class
End Namespace
```

Related rules

CA1059: Members should not expose certain concrete types

CA1012: Abstract types should not have constructors

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1012
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A type is abstract and has a constructor.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

Constructors on abstract types can be called only by derived types. Because public constructors create instances of a type and you cannot create instances of an abstract type, an abstract type that has a public constructor is incorrectly designed.

How to fix violations

To fix a violation of this rule, either make the constructor protected or don't declare the type as abstract.

When to suppress warnings

Do not suppress a warning from this rule. The abstract type has a public constructor.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1012.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following code snippet contains an abstract type that violates this rule.

```
Imports System

Namespace Samples

    ' Violates this rule
    Public MustInherit Class Book

        Public Sub New()
        End Sub

    End Class

End Namespace
```

```
using System;

namespace Samples
{
    // Violates this rule
    public abstract class Book
    {
        public Book()
        {
        }
    }
}
```

The following code snippet fixes the previous violation by changing the accessibility of the constructor from `public` to `protected`.

```
using System;

namespace Samples
{
    // Does not violate this rule
    public abstract class Book
    {
        protected Book()
        {
        }
    }
}
```

```
Imports System

Namespace Samples

    ' Violates this rule
    Public MustInherit Class Book

        Protected Sub New()
        End Sub

    End Class

End Namespace
```

CA1013: Overload operator equals on overloading add and subtract

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1013
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A public or protected type implements the addition or subtraction operators without implementing the equality operator.

Rule description

When instances of a type can be combined by using operations such as addition and subtraction, you should almost always define equality to return `true` for any two instances that have the same constituent values.

You cannot use the default equality operator in an overloaded implementation of the equality operator. Doing so will cause a stack overflow. To implement the equality operator, use the Object.Equals method in your implementation. See the following example.

```
If (Object.ReferenceEquals(left, Nothing)) Then
    Return Object.ReferenceEquals(right, Nothing)
Else
    Return left.Equals(right)
End If
```

```
if (Object.ReferenceEquals(left, null))
    return Object.ReferenceEquals(right, null);
return left.Equals(right);
```

How to fix violations

To fix a violation of this rule, implement the equality operator so that it is mathematically consistent with the addition and subtraction operators.

When to suppress warnings

It is safe to suppress a warning from this rule when the default implementation of the equality operator provides the correct behavior for the type.

Example

The following example defines a type (`BadAddableType`) that violates this rule. This type should implement the equality operator to make any two instances that have the same field values test `true` for equality. The type `GoodAddableType` shows the corrected implementation. Note that this type also implements the inequality operator and overrides Equals to satisfy other rules. A complete implementation would also implement GetHashCode.

```
using System;

namespace DesignLibrary
{
   public class BadAddableType
   {
      private int a, b;
      public BadAddableType(int a, int b)
      {
         this.a = a;
         this.b = b;
      }
      // Violates rule: OverrideOperatorEqualsOnOverridingAddAndSubtract.
      public static BadAddableType operator +(BadAddableType a, BadAddableType b)
      {
         return new BadAddableType(a.a + b.a, a.b + b.b);
      }
      // Violates rule: OverrideOperatorEqualsOnOverridingAddAndSubtract.
      public static BadAddableType operator -(BadAddableType a, BadAddableType b)
      {
         return new BadAddableType(a.a - b.a, a.b - b.b);
      }
      public override string ToString()
      {
         return String.Format("{{{0},{1}}}", a, b);
      }
   }

   public class GoodAddableType
   {
      private int a, b;
      public GoodAddableType(int a, int b)
      {
         this.a = a;
         this.b = b;
      }
      // Satisfies rule: OverrideOperatorEqualsOnOverridingAddAndSubtract.
      public static bool operator ==(GoodAddableType a, GoodAddableType b)
      {
         return (a.a == b.a && a.b == b.b);
      }

      // If you implement ==, you must implement !=.
      public static bool operator !=(GoodAddableType a, GoodAddableType b)
      {
         return !(a==b);
      }

      // Equals should be consistent with operator ==.
      public override bool Equals(Object obj)
      {
         GoodAddableType good = obj as GoodAddableType;
         if (obj == null)
            return false;

        return this == good;
      }

      public static GoodAddableType operator +(GoodAddableType a, GoodAddableType b)
      {
```

```
            return new GoodAddableType(a.a + b.a, a.b + b.b);
        }

        public static GoodAddableType operator -(GoodAddableType a, GoodAddableType b)
        {
            return new GoodAddableType(a.a - b.a, a.b - b.b);
        }
        public override string ToString()
        {
            return String.Format("{{{0},{1}}}", a, b);
        }
    }
}
```

Example

The following example tests for equality by using instances of the types that were previously defined in this topic to illustrate the default and correct behavior for the equality operator.

```
using System;

namespace DesignLibrary
{
    public class TestAddableTypes
    {
        public static void Main()
        {
            BadAddableType a = new BadAddableType(2,2);
            BadAddableType b = new BadAddableType(2,2);
            BadAddableType x = new BadAddableType(9,9);
            GoodAddableType c = new GoodAddableType(3,3);
            GoodAddableType d = new GoodAddableType(3,3);
            GoodAddableType y = new GoodAddableType(9,9);

            Console.WriteLine("Bad type:  {0} {1} are equal? {2}", a,b, a.Equals(b)? "Yes":"No");
            Console.WriteLine("Good type: {0} {1} are equal? {2}", c,d, c.Equals(d)? "Yes":"No");
            Console.WriteLine("Good type: {0} {1} are == ?   {2}", c,d, c==d? "Yes":"No");
            Console.WriteLine("Bad type:  {0} {1} are equal? {2}", a,x, a.Equals(x)? "Yes":"No");
            Console.WriteLine("Good type: {0} {1} are == ?   {2}", c,y, c==y? "Yes":"No");
        }
    }
}
```

This example produces the following output:

```
Bad type:  {2,2} {2,2} are equal? No
Good type: {3,3} {3,3} are equal? Yes
Good type: {3,3} {3,3} are == ?   Yes
Bad type:  {2,2} {9,9} are equal? No
Good type: {3,3} {9,9} are == ?   No
```

See also

- Equality Operators

CA1014: Mark assemblies with CLSCompliantAttribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1014
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

An assembly does not have the System.CLSCompliantAttribute attribute applied to it.

Rule description

The Common Language Specification (CLS) defines naming restrictions, data types, and rules to which assemblies must conform if they will be used across programming languages. Good design dictates that all assemblies explicitly indicate CLS compliance with CLSCompliantAttribute. If the attribute is not present on an assembly, the assembly is not compliant.

It is possible for a CLS-compliant assembly to contain types or type members that are not compliant.

How to fix violations

To fix a violation of this rule, add the attribute to the assembly. Instead of marking the whole assembly as noncompliant, you should determine which type or type members are not compliant and mark these elements as such. If possible, you should provide a CLS-compliant alternative for noncompliant members so that the widest possible audience can access all the functionality of your assembly.

When to suppress warnings

Do not suppress a warning from this rule. If you do not want the assembly to be compliant, apply the attribute and set its value to `false`.

Example

The following example shows an assembly that has the System.CLSCompliantAttribute attribute applied that declares it CLS-compliant.

```
using System;

[assembly:CLSCompliant(true)]
namespace DesignLibrary {}
```

```
using namespace System;

[assembly:CLSCompliant(true)];
namespace DesignLibrary {}
```

```
Imports System

<assembly:CLSCompliant(true)>
Namespace DesignLibrary
End Namespace
```

See also

- System.CLSCompliantAttribute
- Language Independence and Language-Independent Components

CA1016: Mark assemblies with AssemblyVersionAttribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1016
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

The assembly does not have a version number.

Rule description

The identity of an assembly is composed of the following information:

- Assembly name
- Version number
- Culture
- Public key (for strongly named assemblies).

.NET uses the version number to uniquely identify an assembly and to bind to types in strongly named assemblies. The version number is used together with version and publisher policy. By default, applications run only with the assembly version with which they were built.

How to fix violations

To fix a violation of this rule, add a version number to the assembly by using the System.Reflection.AssemblyVersionAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule for assemblies that are used by third parties or in a production environment.

Example

The following example shows an assembly that has the AssemblyVersionAttribute attribute applied.

```
using System;
using System.Reflection;

[assembly: AssemblyVersionAttribute("4.3.2.1")]
namespace DesignLibrary {}
```

```
Imports System
Imports System.Reflection

<Assembly: AssemblyVersionAttribute("4.3.2.1")>
Namespace DesignLibrary
End Namespace
```

```
using namespace System;
using namespace System::Reflection;

[assembly: AssemblyVersionAttribute("4.3.2.1")];
namespace DesignLibrary {}
```

See also

- Assembly versioning
- How to: Create a publisher policy

CA1017: Mark assemblies with ComVisibleAttribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1017
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

An assembly does not have the System.Runtime.InteropServices.ComVisibleAttribute attribute applied to it.

Rule description

The ComVisibleAttribute attribute determines how COM clients access managed code. Good design dictates that assemblies explicitly indicate COM visibility. COM visibility can be set for a whole assembly and then overridden for individual types and type members. If the attribute is not present, the contents of the assembly are visible to COM clients.

How to fix violations

To fix a violation of this rule, add the attribute to the assembly. If you do not want the assembly to be visible to COM clients, apply the attribute and set its value to `false`.

When to suppress warnings

Do not suppress a warning from this rule. If you want the assembly to be visible, apply the attribute and set its value to `true`.

Example

The following example shows an assembly that has the ComVisibleAttribute attribute applied to prevent it from being visible to COM clients.

```
using namespace System;

[assembly: System::Runtime::InteropServices::ComVisible(false)];
namespace DesignLibrary {}
```

```
Imports System

<Assembly: System.Runtime.InteropServices.ComVisible(False)>
Namespace DesignLibrary
End Namespace
```

```
using System;

[assembly: System.Runtime.InteropServices.ComVisible(false)]
namespace DesignLibrary {}
```

See also

- Interoperating with Unmanaged Code
- Qualifying .NET Types for Interoperation

CA1018: Mark attributes with AttributeUsageAttribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1018
Category	Microsoft.Design
Breaking change	Breaking

Cause

The System.AttributeUsageAttribute attribute is not present on the custom attribute.

Rule description

When you define a custom attribute, mark it by using AttributeUsageAttribute to indicate where in the source code the custom attribute can be applied. The meaning and intended usage of an attribute will determine its valid locations in code. For example, you might define an attribute that identifies the person who is responsible for maintaining and enhancing each type in a library, and that responsibility is always assigned at the type level. In this case, compilers should enable the attribute on classes, enumerations, and interfaces, but should not enable it on methods, events, or properties. Organizational policies and procedures would dictate whether the attribute should be enabled on assemblies.

The System.AttributeTargets enumeration defines the targets that you can specify for a custom attribute. If you omit AttributeUsageAttribute, your custom attribute will be valid for all targets, as defined by the `All` value of AttributeTargets enumeration.

How to fix violations

To fix a violation of this rule, specify targets for the attribute by using AttributeUsageAttribute. See the following example.

When to suppress warnings

You should fix a violation of this rule instead of excluding the message. Even if the attribute inherits AttributeUsageAttribute, the attribute should be present to simplify code maintenance.

Example

The following example defines two attributes. `BadCodeMaintainerAttribute` incorrectly omits the AttributeUsageAttribute statement, and `GoodCodeMaintainerAttribute` correctly implements the attribute that is described earlier in this section. Note that the property `DeveloperName` is required by the design rule CA1019: Define accessors for attribute arguments and is included for completeness.

```
using System;

namespace DesignLibrary
{
// Violates rule: MarkAttributesWithAttributeUsage.

   public sealed class BadCodeMaintainerAttribute :Attribute
   {
      string developer;

      public BadCodeMaintainerAttribute(string developerName)
      {
         developer = developerName;
      }
      public string DeveloperName
      {
         get
         {
            return developer;
         }
      }
   }
// Satisfies rule: Attributes specify AttributeUsage.

   // The attribute is valid for type-level targets.
   [AttributeUsage(AttributeTargets.Class | AttributeTargets.Enum | AttributeTargets.Interface |
AttributeTargets.Delegate)]
   public sealed class GoodCodeMaintainerAttribute :Attribute
   {
      string developer;

      public GoodCodeMaintainerAttribute(string developerName)
      {
         developer = developerName;
      }
      public string DeveloperName
      {
         get
         {
            return developer;
         }
      }
   }
}
```

```
Imports System

Namespace DesignLibrary

' Violates rule: MarkAttributesWithAttributeUsage.
NotInheritable Public Class BadCodeMaintainerAttribute
    Inherits Attribute
    Private developer As String

    Public Sub New(developerName As String)
        developer = developerName
    End Sub 'New

    Public ReadOnly Property DeveloperName() As String
        Get
            Return developer
        End Get
    End Property
End Class

' Satisfies rule: Attributes specify AttributeUsage.
' The attribute is valid for type-level targets.
<AttributeUsage(AttributeTargets.Class Or AttributeTargets.Enum Or _
   AttributeTargets.Interface Or AttributeTargets.Delegate)> _
NotInheritable Public Class GoodCodeMaintainerAttribute
    Inherits Attribute
    Private developer As String

    Public Sub New(developerName As String)
        developer = developerName
    End Sub 'New

    Public ReadOnly Property DeveloperName() As String
        Get
            Return developer
        End Get
    End Property
End Class

End Namespace
```

Related rules

CA1019: Define accessors for attribute arguments

CA1813: Avoid unsealed attributes

See also

- Attributes

CA1019: Define accessors for attribute arguments

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1019
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

In its constructor, an attribute defines arguments that do not have corresponding properties.

Rule description

Attributes can define mandatory arguments that must be specified when you apply the attribute to a target. These are also known as positional arguments because they are supplied to attribute constructors as positional parameters. For every mandatory argument, the attribute should also provide a corresponding read-only property so that the value of the argument can be retrieved at execution time. This rule checks that for each constructor parameter, you have defined the corresponding property.

Attributes can also define optional arguments, which are also known as named arguments. These arguments are supplied to attribute constructors by name and should have a corresponding read/write property.

For mandatory and optional arguments, the corresponding properties and constructor parameters should use the same name but different casing. Properties use Pascal casing, and parameters use camel casing.

How to fix violations

To fix a violation of this rule, add a read-only property for each constructor parameter that does not have one.

When to suppress warnings

Suppress a warning from this rule if you do not want the value of the mandatory argument to be retrievable.

Custom Attributes Example

The following example shows two attributes that define a mandatory (positional) parameter. The first implementation of the attribute is incorrectly defined. The second implementation is correct.

```
using System;

namespace DesignLibrary
{
// Violates rule: DefineAccessorsForAttributeArguments.

   [AttributeUsage(AttributeTargets.All)]
   public sealed class BadCustomAttribute :Attribute
   {
      string data;

      // Missing the property that corresponds to
      // the someStringData parameter.

      public BadCustomAttribute(string someStringData)
      {
         data = someStringData;
      }
   }

// Satisfies rule: Attributes should have accessors for all arguments.

   [AttributeUsage(AttributeTargets.All)]
   public sealed class GoodCustomAttribute :Attribute
   {
      string data;

      public GoodCustomAttribute(string someStringData)
      {
         data = someStringData;
      }
      //The constructor parameter and property
      //name are the same except for case.

      public string SomeStringData
      {
         get
         {
            return data;
         }
      }
   }
}
```

```
Imports System

Namespace DesignLibrary

' Violates rule: DefineAccessorsForAttributeArguments.
<AttributeUsage(AttributeTargets.All)>  _
NotInheritable Public Class BadCustomAttribute
    Inherits Attribute
    Private data As String

    ' Missing the property that corresponds to
    ' the someStringData parameter.
    Public Sub New(someStringData As String)
        data = someStringData
    End Sub 'New
End Class 'BadCustomAttribute

' Satisfies rule: Attributes should have accessors for all arguments.
<AttributeUsage(AttributeTargets.All)>  _
NotInheritable Public Class GoodCustomAttribute
    Inherits Attribute
    Private data As String

    Public Sub New(someStringData As String)
        data = someStringData
    End Sub 'New

    'The constructor parameter and property
    'name are the same except for case.

    Public ReadOnly Property SomeStringData() As String
        Get
            Return data
        End Get
    End Property
End Class

End Namespace
```

Positional and Named Arguments

Positional and named arguments make it clear to consumers of your library which arguments are mandatory for the attribute and which arguments are optional.

The following example shows an implementation of an attribute that has both positional and named arguments:

```
using System;

namespace DesignLibrary
{
    [AttributeUsage(AttributeTargets.All)]
    public sealed class GoodCustomAttribute : Attribute
    {
        string mandatory;
        string optional;

        public GoodCustomAttribute(string mandatoryData)
        {
            mandatory = mandatoryData;
        }

        public string MandatoryData
        {
            get { return mandatory; }
        }

        public string OptionalData
        {
            get { return optional; }
            set { optional = value; }
        }
    }
}
```

The following example shows how to apply the custom attribute to two properties:

```
[GoodCustomAttribute("ThisIsSomeMandatoryData", OptionalData = "ThisIsSomeOptionalData")]
public string MyProperty
{
    get { return myProperty; }
    set { myProperty = value; }
}

[GoodCustomAttribute("ThisIsSomeMoreMandatoryData")]
public string MyOtherProperty
{
    get { return myOtherProperty; }
    set { myOtherProperty = value; }
}
```

Related rules

CA1813: Avoid unsealed attributes

See also

Attributes

CA1020: Avoid namespaces with few types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1020
Category	Microsoft.Design
Breaking change	Breaking

Cause

A namespace other than the global namespace contains fewer than five types.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Make sure that each of your namespaces has a logical organization, and that a valid reason exists to put types in a sparsely populated namespace. Namespaces should contain types that are used together in most scenarios. When their applications are mutually exclusive, types should be located in separate namespaces. For example, the System.Web.UI namespace contains types that are used in web applications, and the System.Windows.Forms namespace contains types that are used in Windows-based applications. Even though both namespaces have types that control aspects of the user interface, these types are not designed for use in the same application. Therefore, they are located in separate namespaces. Careful namespace organization can also be helpful because it increases the discoverability of a feature. By examining the namespace hierarchy, library consumers should be able to locate the types that implement a feature.

> **NOTE**
> Design-time types and permissions should not be merged into other namespaces to comply with this guideline. These types belong in their own namespaces below your main namespace, and the namespaces should end in `.Design` and `.Permissions`, respectively.

How to fix violations

To fix a violation of this rule, try to combine namespaces that contain just a few types into a single namespace.

When to suppress warnings

It is safe to suppress a warning from this rule when the namespace does not contain types that are used with the types in your other namespaces.

CA1021: Avoid out parameters

8/4/2020 • 7 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1021
Category	Microsoft.Design
Breaking change	Breaking

Cause

A public or protected method in a public type has an `out` parameter.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

Passing types by reference (using `out` or `ref`) requires experience with pointers, understanding how value types and reference types differ, and handling methods with multiple return values. Also, the difference between `out` and `ref` parameters is not widely understood.

When a reference type is passed "by reference," the method intends to use the parameter to return a different instance of the object. Passing a reference type by reference is also known as using a double pointer, pointer to a pointer, or double indirection. By using the default calling convention, which is pass "by value," a parameter that takes a reference type already receives a pointer to the object. The pointer, not the object to which it points, is passed by value. Pass by value means that the method cannot change the pointer to have it point to a new instance of the reference type. However, it can change the contents of the object to which it points. For most applications this is sufficient and yields the desired behavior.

If a method must return a different instance, use the return value of the method to accomplish this. See the System.String class for a variety of methods that operate on strings and return a new instance of a string. When this model is used, the caller must decide whether the original object is preserved.

Although return values are commonplace and heavily used, the correct application of `out` and `ref` parameters requires intermediate design and coding skills. Library architects who design for a general audience should not expect users to master working with `out` or `ref` parameters.

How to fix violations

To fix a violation of this rule that is caused by a value type, have the method return the object as its return value. If the method must return multiple values, redesign it to return a single instance of an object that holds the values.

To fix a violation of this rule that is caused by a reference type, make sure that the desired behavior is to return a new instance of the reference. If it is, the method should use its return value to do this.

When to suppress warnings

It is safe to suppress a warning from this rule. However, this design could cause usability issues.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1021.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following library shows two implementations of a class that generates responses to the feedback of a user. The first implementation (`BadRefAndOut`) forces the library user to manage three return values. The second implementation (`RedesignedRefAndOut`) simplifies the user experience by returning an instance of a container class (`ReplyData`) that manages the data as a single unit.

```
using System;

namespace DesignLibrary
{
   public enum Actions
   {
      Unknown,
      Discard,
      ForwardToManagement,
      ForwardToDeveloper
   }

   public enum TypeOfFeedback
   {
      Complaint,
      Praise,
      Suggestion,
      Incomprehensible
   }

   public class BadRefAndOut
   {
      // Violates rule: DoNotPassTypesByReference.

      public static bool ReplyInformation (TypeOfFeedback input,
         out string reply, ref Actions action)
      {
         bool returnReply = false;
         string replyText = "Your feedback has been forwarded " +
                            "to the product manager.";

         reply = String.Empty;
         switch (input)
         {
            case TypeOfFeedback.Complaint:
            case TypeOfFeedback.Praise :
               action = Actions.ForwardToManagement;
               reply = "Thank you. " + replyText;
               returnReply = true;
               break;
            case TypeOfFeedback.Suggestion:
               action = Actions.ForwardToDeveloper;
               reply = replyText;
               returnReply = true;
               break;
```

```
            break;
         case TypeOfFeedback.Incomprehensible:
         default:
            action = Actions.Discard;
            returnReply = false;
            break;
      }
      return returnReply;
   }
}

// Redesigned version does not use out or ref parameters;
// instead, it returns this container type.

public class ReplyData
{
   string reply;
   Actions action;
   bool returnReply;

   // Constructors.
   public ReplyData()
   {
      this.reply = String.Empty;
      this.action = Actions.Discard;
      this.returnReply = false;
   }

   public ReplyData (Actions action, string reply, bool returnReply)
   {
      this.reply = reply;
      this.action = action;
      this.returnReply = returnReply;
   }

   // Properties.
   public string Reply { get { return reply;}}
   public Actions Action { get { return action;}}

   public override string ToString()
   {
      return String.Format("Reply: {0} Action: {1} return? {2}",
         reply, action.ToString(), returnReply.ToString());
   }
}

public class RedesignedRefAndOut
{
   public static ReplyData ReplyInformation (TypeOfFeedback input)
   {
      ReplyData answer;
      string replyText = "Your feedback has been forwarded " +
         "to the product manager.";

      switch (input)
      {
         case TypeOfFeedback.Complaint:
         case TypeOfFeedback.Praise :
            answer = new ReplyData(
               Actions.ForwardToManagement,
               "Thank you. " + replyText,
               true);
            break;
         case TypeOfFeedback.Suggestion:
            answer =  new ReplyData(
               Actions.ForwardToDeveloper,
               replyText,
               true);
            break;
```

```
            case TypeOfFeedback.Incomprehensible:
            default:
               answer = new ReplyData();
               break;
         }
         return answer;
      }
   }
}
```

Example

The following application illustrates the experience of the user. The call to the redesigned library (`UseTheSimplifiedClass` method) is more straightforward, and the information returned by the method is easily managed. The output from the two methods is identical.

```
using System;

namespace DesignLibrary
{
   public class UseComplexMethod
   {
      static void UseTheComplicatedClass()
      {
         // Using the version with the ref and out parameters.
         // You do not have to initialize an out parameter.

         string[] reply = new string[5];

         // You must initialize a ref parameter.
         Actions[] action = {Actions.Unknown,Actions.Unknown,
                             Actions.Unknown,Actions.Unknown,
                             Actions.Unknown,Actions.Unknown};
         bool[] disposition= new bool[5];
         int i = 0;

         foreach(TypeOfFeedback t in Enum.GetValues(typeof(TypeOfFeedback)))
         {
            // The call to the library.
            disposition[i] = BadRefAndOut.ReplyInformation(
               t, out reply[i], ref action[i]);
            Console.WriteLine("Reply: {0} Action: {1}  return? {2} ",
               reply[i], action[i], disposition[i]);
            i++;
         }
      }

      static void UseTheSimplifiedClass()
      {
         ReplyData[] answer = new ReplyData[5];
         int i = 0;
         foreach(TypeOfFeedback t in Enum.GetValues(typeof(TypeOfFeedback)))
         {
            // The call to the library.
            answer[i] = RedesignedRefAndOut.ReplyInformation(t);
            Console.WriteLine(answer[i++]);
         }
      }

      public  static void Main()
      {
         UseTheComplicatedClass();

         // Print a blank line in output.
         Console.WriteLine("");

         UseTheSimplifiedClass();
      }
   }
}
```

Example

The following example library illustrates how `ref` parameters for reference types are used and shows a better way to implement this functionality.

```
using System;

namespace DesignLibrary
{
   public class ReferenceTypesAndParameters
   {

      // The following syntax will not work. You cannot make a
      // reference type that is passed by value point to a new
      // instance. This needs the ref keyword.

      public static void BadPassTheObject(string argument)
      {
         argument = argument + " ABCDE";
      }

      // The following syntax will work, but is considered bad design.
      // It reassigns the argument to point to a new instance of string.
      // Violates rule DoNotPassTypesByReference.

      public static void PassTheReference(ref string argument)
      {
         argument = argument + " ABCDE";
      }

      // The following syntax will work and is a better design.
      // It returns the altered argument as a new instance of string.

      public static string BetterThanPassTheReference(string argument)
      {
         return argument + " ABCDE";
      }
   }
}
```

Example

The following application calls each method in the library to demonstrate the behavior.

```
using System;

namespace DesignLibrary
{
   public class Test
   {
      public static void Main()
      {
         string s1 = "12345";
         string s2 = "12345";
         string s3 = "12345";

         Console.WriteLine("Changing pointer - passed by value:");
         Console.WriteLine(s1);
         ReferenceTypesAndParameters.BadPassTheObject (s1);
         Console.WriteLine(s1);

         Console.WriteLine("Changing pointer - passed by reference:");
         Console.WriteLine(s2);
         ReferenceTypesAndParameters.PassTheReference (ref s2);
         Console.WriteLine(s2);

         Console.WriteLine("Passing by return value:");
         s3 = ReferenceTypesAndParameters.BetterThanPassTheReference (s3);
         Console.WriteLine(s3);
      }
   }
}
```

This example produces the following output:

```
Changing pointer - passed by value:
12345
12345
Changing pointer - passed by reference:
12345
12345 ABCDE
Passing by return value:
12345 ABCDE
```

Try pattern methods

Description

Methods that implement the `Try<Something>` pattern, such as System.Int32.TryParse, do not raise this violation. The following example shows a structure (value type) that implements the System.Int32.TryParse method.

Code

```
using System;

namespace Samples
{
    public struct Point
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int axisX, int axisY)
        {
            _X = axisX;
            _Y = axisY;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override int GetHashCode()
        {
            return _X ^ _Y;
        }

        public override bool Equals(object obj)
        {
            if (!(obj is Point))
                return false;

            return Equals((Point)obj);
        }

        public bool Equals(Point other)
        {
            if (_X != other._X)
                return false;

            return _Y == other._Y;
        }

        public static bool operator ==(Point point1, Point point2)
        {
            return point1.Equals(point2);
        }

        public static bool operator !=(Point point1, Point point2)
        {
            return !point1.Equals(point2);
        }

        // Does not violate this rule
        public static bool TryParse(string value, out Point result)
        {
            // TryParse Implementation
            result = new Point(0,0);
            return false;
        }
    }
}
```

Related rules

CA1045: Do not pass types by reference

CA1023: Indexers should not be multidimensional

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1023
Category	Microsoft.Design
Breaking change	Breaking

Cause

A public or protected type contains a public or protected indexer that uses more than one index.

Rule description

Indexers, that is, indexed properties, should use a single index. Multi-dimensional indexers can significantly reduce the usability of the library. If the design requires multiple indexes, reconsider whether the type represents a logical data store. If not, use a method.

How to fix violations

To fix a violation of this rule, change the design to use a lone integer or string index, or use a method instead of the indexer.

When to suppress warnings

Suppress a warning from this rule only after carefully considering the need for the nonstandard indexer.

Example

The following example shows a type, `DayOfWeek03`, with a multi-dimensional indexer that violates the rule. The indexer can be seen as a type of conversion and therefore is more appropriately exposed as a method. The type is redesigned in `RedesignedDayOfWeek03` to satisfy the rule.

```
Imports System

Namespace DesignLibrary

    Public Class DayOfWeek03

        Private dayOfWeek(,) As String = {{"Wed", "Thu", "..."}, _
                                          {"Sat", "Sun", "..."}}
                                          ' ...

        Default ReadOnly Property Item(month As Integer, day As Integer) As String
            Get
                Return dayOfWeek(month - 1, day - 1)
            End Get
        End Property

    End Class

    Public Class RedesignedDayOfWeek03

        Private dayOfWeek() As String = _
            {"Tue", "Wed", "Thu", "Fri", "Sat", "Sun", "Mon"}
        Private daysInPreviousMonth() As Integer = _
            {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30}

        Function GetDayOfWeek(month As Integer, day As Integer) As String
            Return dayOfWeek((daysInPreviousMonth(month - 1) + day) Mod 7)
        End Function

    End Class

End Namespace
```

```
using namespace System;

namespace DesignLibrary
{
    public ref class DayOfWeek03
    {
        array<String^, 2>^ dayOfWeek;

    public:
        property String^ default[int,  int]
        {
            String^ get(int month, int day)
            {
                return dayOfWeek[month - 1, day - 1];
            }
        }

        DayOfWeek03()
        {
            dayOfWeek = gcnew array<String^, 2>(12, 7);
            dayOfWeek[0,0] = "Wed";
            dayOfWeek[0,1] = "Thu";
            // ...
            dayOfWeek[1,0] = "Sat";
            dayOfWeek[1,1] = "Sun";
            // ...
        }
    };

    public ref class RedesignedDayOfWeek03
    {
        static array<String^>^ dayOfWeek =
            {"Tue", "Wed", "Thu", "Fri", "Sat", "Sun", "Mon"};

        static array<int>^ daysInPreviousMonth =
            {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30};

    public:
        String^ GetDayOfWeek(int month, int day)
        {
            return dayOfWeek[(daysInPreviousMonth[month - 1] + day) % 7];
        }
    };
}
```

```
using System;

namespace DesignLibrary
{
    public class DayOfWeek03
    {
        string[,] dayOfWeek = {{"Wed", "Thu", "..."},
                               {"Sat", "Sun", "..."}};
                               // ...

        public string this[int month, int day]
        {
            get
            {
                return dayOfWeek[month - 1, day - 1];
            }
        }
    }

    public class RedesignedDayOfWeek03
    {
        string[] dayOfWeek =
            {"Tue", "Wed", "Thu", "Fri", "Sat", "Sun", "Mon"};

        int[] daysInPreviousMonth =
            {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30};

        public string GetDayOfWeek(int month, int day)
        {
            return dayOfWeek[(daysInPreviousMonth[month - 1] + day) % 7];
        }
    }
}
```

Related rules

CA1043: Use integral or string argument for indexers

CA1024: Use properties where appropriate

CA1024: Use properties where appropriate

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1024
Category	Microsoft.Design
Breaking change	Breaking

Cause

A method has a name that starts with `Get`, takes no parameters, and returns a value that is not an array.

By default, this rule only looks at public and protected methods, but this is configurable.

Rule description

In most cases, properties represent data and methods perform actions. Properties are accessed like fields, which makes them easier to use. A method is a good candidate to become a property if one of these conditions is present:

- Takes no arguments and returns the state information of an object.
- Accepts a single argument to set some part of the state of an object.

Properties should behave as if they are fields; if the method cannot, it should not be changed to a property. Methods are better than properties in the following situations:

- The method performs a time-consuming operation. The method is perceivably slower than the time that is required to set or get the value of a field.
- The method performs a conversion. Accessing a field does not return a converted version of the data that it stores.
- The Get method has an observable side effect. Retrieving the value of a field does not produce any side effects.
- The order of execution is important. Setting the value of a field does not rely on the occurrence of other operations.
- Calling the method two times in succession creates different results.
- The method is static but returns an object that can be changed by the caller. Retrieving the value of a field does not allow the caller to change the data that is stored by the field.
- The method returns an array.

How to fix violations

To fix a violation of this rule, change the method to a property.

When to suppress warnings

Suppress a warning from this rule if the method meets at least one of the previously listed criteria.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1024.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Control property expansion in the debugger

One reason programmers avoid using a property is because they do not want the debugger to autoexpand it. For example, the property might involve allocating a large object or calling a P/Invoke, but it might not actually have any observable side effects.

You can prevent the debugger from autoexpanding properties by applying System.Diagnostics.DebuggerBrowsableAttribute. The following example shows this attribute being applied to an instance property.

```
Imports System
Imports System.Diagnostics

Namespace Microsoft.Samples

    Public Class TestClass

        ' [...]

        <DebuggerBrowsable(DebuggerBrowsableState.Never)> _
        Public ReadOnly Property LargeObject() As LargeObject
            Get
                ' Allocate large object
                ' [...]
            End Get
        End Property

    End Class

End Namespace
```

```
using System;
using System.Diagnostics;

namespace Microsoft.Samples
{
    publicclass TestClass
    {
        // [...]

        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public LargeObject LargeObject
        {
            get
            {
                // Allocate large object
                // [...]
            }
        }
    }
}
```

Example

The following example contains several methods that should be converted to properties and several that should not because they don't behave like fields.

```
using System;
using System.Globalization;
using System.Collections;
namespace DesignLibrary
{
   // Illustrates the behavior of rule:
   //  UsePropertiesWhereAppropriate.

   public class Appointment
   {
      static long nextAppointmentID;
      static double[] discountScale = {5.0, 10.0, 33.0};
      string customerName;
      long customerID;
      DateTime when;

      // Static constructor.
      static Appointment()
      {
         // Initializes the static variable for Next appointment ID.
      }

      // This method will violate the rule, but should not be a property.
      // This method has an observable side effect.
      // Calling the method twice in succession creates different results.
      public static long GetNextAvailableID()
      {
         nextAppointmentID++;
         return nextAppointmentID - 1;
      }

      // This method will violate the rule, but should not be a property.
      // This method performs a time-consuming operation.
      // This method returns an array.

      public Appointment[] GetCustomerHistory()
      {
         // Connect to a database to get the customer's appointment history.
         return LoadHistoryFromDB(customerID);
```

```
        return LoadHistoryFromDB(customerID);
      }

      // This method will violate the rule, but should not be a property.
      // This method is static but returns a mutable object.
      public static double[] GetDiscountScaleForUpdate()
      {
         return discountScale;
      }

      // This method will violate the rule, but should not be a property.
      // This method performs a conversion.
      public string GetWeekDayString()
      {
         return DateTimeFormatInfo.CurrentInfo.GetDayName(when.DayOfWeek);
      }

      // These methods will violate the rule, and should be properties.
      // They each set or return a piece of the current object's state.

      public DayOfWeek GetWeekDay ()
      {
         return when.DayOfWeek;
      }

      public void  SetCustomerName (string customerName)
      {
         this.customerName = customerName;
      }
      public string GetCustomerName ()
      {
         return customerName;
      }

     public void SetCustomerID (long customerID)
      {
         this.customerID = customerID;
      }

      public long GetCustomerID ()
      {
         return customerID;
      }

      public void SetScheduleTime (DateTime when)
      {
         this.when = when;
      }

      public DateTime GetScheduleTime ()
      {
         return when;
      }

      // Time-consuming method that is called by GetCustomerHistory.
      Appointment[] LoadHistoryFromDB(long customerID)
      {
         ArrayList records = new ArrayList();
         // Load from database.
         return (Appointment[])records.ToArray();
      }
   }
}
```

CA1025: Replace repetitive arguments with params array

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1025
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A public or protected method in a public type has more than three parameters, and its last three parameters are the same type.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Use a parameter array instead of repeated arguments when the exact number of arguments is unknown and the variable arguments are the same type, or can be passed as the same type. For example, the WriteLine method provides a general-purpose overload that uses a parameter array to accept any number of Object arguments.

How to fix violations

To fix a violation of this rule, replace the repeated arguments with a parameter array.

When to suppress warnings

It is always safe to suppress a warning from this rule; however, this design might cause usability issues.

Example

The following example shows a type that violates this rule.

```
using System;

namespace DesignLibrary
{
   public class BadRepeatArguments
   {
      // Violates rule: ReplaceRepetitiveArgumentsWithParamsArray.
      public void VariableArguments(object obj1, object obj2, object obj3, object obj4) {}
      public void VariableArguments(object obj1, object obj2, object obj3, object obj4, object obj5) {}
   }

   public class GoodRepeatArguments
   {
       public void VariableArguments(object obj1) {}
       public void VariableArguments(object obj1, object obj2) {}
       public void VariableArguments(object obj1, object obj2, object obj3) {}
       public void VariableArguments(params Object[] arg) {}
   }
}
```

CA1026: Default parameters should not be used

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1026
Category	Microsoft.Design
Breaking change	Breaking

Cause

An externally visible type contains an externally visible method that uses a default parameter.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Methods that use default parameters are allowed under the Common Language Specification (CLS); however, the CLS allows compilers to ignore the values that are assigned to these parameters. Code that is written for compilers that ignore default parameter values must explicitly provide arguments for each default parameter. To maintain the behavior that you want across programming languages, methods that use default parameters should be replaced with method overloads that provide the default parameters.

The compiler ignores the values of default parameters for Managed Extension for C++ when it accesses managed code. The Visual Basic compiler supports methods that have default parameters that use the Optional keyword.

How to fix violations

To fix a violation of this rule, replace the method that uses default parameters with method overloads that supply the default parameters.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a method that uses default parameters, and the overloaded methods that provide an equivalent functionality.

```
Imports System

<Assembly: CLSCompliant(True)>
Namespace DesignLibrary

    Public Class DefaultVersusOverloaded

        Sub DefaultParameters(Optional parameter1 As Integer = 1, _
                              Optional parameter2 As Integer = 5)
            ' ...
            Console.WriteLine("{0} : {1}", parameter1, parameter2)
        End Sub

        Sub OverloadedMethod()
            OverloadedMethod(1, 5)
        End Sub

        Sub OverloadedMethod(parameter1 As Integer)
            OverloadedMethod(parameter1, 5)
        End Sub

        Sub OverloadedMethod(parameter1 As Integer, parameter2 As Integer)
            ' ...
            Console.WriteLine("{0} : {1}", parameter1, parameter2)
        End Sub

    End Class

End Namespace
```

Related rules

CA1025: Replace repetitive arguments with params array

See also

Language Independence and Language-Independent Components

CA1027: Mark enums with FlagsAttribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1027
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

The values of an enumeration are powers of two or are combinations of other values that are defined in the enumeration, and the System.FlagsAttribute attribute is not present. To reduce false positives, this rule does not report a violation for enumerations that have contiguous values.

By default, this rule only looks at public enumerations, but this is configurable.

Rule description

An enumeration is a value type that defines a set of related named constants. Apply FlagsAttribute to an enumeration when its named constants can be meaningfully combined. For example, consider an enumeration of the days of the week in an application that keeps track of which day's resources are available. If the availability of each resource is encoded by using the enumeration that has FlagsAttribute present, any combination of days can be represented. Without the attribute, only one day of the week can be represented.

For fields that store combinable enumerations, the individual enumeration values are treated as groups of bits in the field. Therefore, such fields are sometimes referred to as *bit fields*. To combine enumeration values for storage in a bit field, use the Boolean conditional operators. To test a bit field to determine whether a specific enumeration value is present, use the Boolean logical operators. For a bit field to store and retrieve combined enumeration values correctly, each value that is defined in the enumeration must be a power of two. Unless this is so, the Boolean logical operators will not be able to extract the individual enumeration values that are stored in the field.

How to fix violations

To fix a violation of this rule, add FlagsAttribute to the enumeration.

When to suppress warnings

Suppress a warning from this rule if you do not want the enumeration values to be combinable.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1027.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

In the following example, `DaysEnumNeedsFlags` is an enumeration that meets the requirements for using FlagsAttribute but doesn't have it. The `ColorEnumShouldNotHaveFlag` enumeration does not have values that are powers of two but incorrectly specifies FlagsAttribute. This violates rule CA2217: Do not mark enums with FlagsAttribute.

```
using System;

namespace DesignLibrary
{
// Violates rule: MarkEnumsWithFlags.

   public enum DaysEnumNeedsFlags
   {
      None        = 0,
      Monday      = 1,
      Tuesday     = 2,
      Wednesday   = 4,
      Thursday    = 8,
      Friday      = 16,
      All         = Monday| Tuesday | Wednesday | Thursday | Friday
   }
   // Violates rule: DoNotMarkEnumsWithFlags.
   [FlagsAttribute]
   public enum ColorEnumShouldNotHaveFlag
   {
      None        = 0,
      Red         = 1,
      Orange      = 3,
      Yellow      = 4
   }
}
```

Related rules

- CA2217: Do not mark enums with FlagsAttribute

See also

- System.FlagsAttribute

CA1028: Enum storage should be Int32

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1028
Category	Microsoft.Design
Breaking change	Breaking

Cause

The underlying type of an enumeration is not System.Int32.

By default, this rule only looks at public enumerations, but this is configurable.

Rule description

An enumeration is a value type that defines a set of related named constants. By default, the System.Int32 data type is used to store the constant value. Even though you can change this underlying type, it is not necessary or recommended for most scenarios. No significant performance gain is achieved by using a data type that is smaller than Int32. If you cannot use the default data type, you should use one of the Common Language System (CLS)-compliant integral types, Byte, Int16, Int32, or Int64 to make sure that all values of the enumeration can be represented in CLS-compliant programming languages.

How to fix violations

To fix a violation of this rule, unless size or compatibility issues exist, use Int32. For situations where Int32 is not large enough to hold the values, use Int64. If backward compatibility requires a smaller data type, use Byte or Int16.

When to suppress warnings

Suppress a warning from this rule only if backward compatibility issues require it. In applications, failure to comply with this rule usually does not cause problems. In libraries, where language interoperability is required, failure to comply with this rule might adversely affect your users.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1028.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example of a violation

The following example shows two enumerations that don't use the recommended underlying data type.

```
Imports System

Namespace Samples

    <Flags()> _
    Public Enum Days As UInteger
        None = 0
        Monday = 1
        Tuesday = 2
        Wednesday = 4
        Thursday = 8
        Friday = 16
        All = Monday Or Tuesday Or Wednesday Or Thursday Or Friday
    End Enum

    Public Enum Color As SByte
        None = 0
        Red = 1
        Orange = 3
        Yellow = 4
    End Enum

End Namespace
```

```
using System;

namespace DesignLibrary
{
   [Flags]
   public enum Days : uint
   {
      None        = 0,
      Monday      = 1,
      Tuesday     = 2,
      Wednesday   = 4,
      Thursday    = 8,
      Friday      = 16,
      All         = Monday| Tuesday | Wednesday | Thursday | Friday
   }

   public enum Color :sbyte
   {
      None        = 0,
      Red         = 1,
      Orange      = 3,
      Yellow      = 4
   }
}
```

Example of how to fix

The following example fixes the previous violation by changing the underlying data type to Int32.

```
using System;

namespace Samples
{
    [Flags]
    public enum Days : int
    {
        None        = 0,
        Monday      = 1,
        Tuesday     = 2,
        Wednesday   = 4,
        Thursday    = 8,
        Friday      = 16,
        All         = Monday| Tuesday | Wednesday | Thursday | Friday
    }

    public enum Color : int
    {
        None        = 0,
        Red         = 1,
        Orange      = 3,
        Yellow      = 4
    }
}
```

```
Imports System

Namespace Samples

    <Flags()> _
    Public Enum Days As Integer
        None = 0
        Monday = 1
        Tuesday = 2
        Wednesday = 4
        Thursday = 8
        Friday = 16
        All = Monday Or Tuesday Or Wednesday Or Thursday Or Friday
    End Enum

    Public Enum Color As Integer
        None = 0
        Red = 1
        Orange = 3
        Yellow = 4
    End Enum

End Namespace
```

Related rules

- CA1008: Enums should have zero value
- CA1027: Mark enums with FlagsAttribute
- CA2217: Do not mark enums with FlagsAttribute
- CA1700: Do not name enum values 'Reserved'
- CA1712: Do not prefix enum values with type name

See also

- System.Byte

- System.Int16
- System.Int32
- System.Int64

CA1030: Use events where appropriate

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1030
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A method name begins with one of the following:

- AddOn
- RemoveOn
- Fire
- Raise

By default, this rule only looks at externally visible methods, but this is configurable.

Rule description

This rule detects methods that have names that ordinarily would be used for events. Events follow the Observer or Publish-Subscribe design pattern; they are used when a state change in one object must be communicated to other objects. If a method gets called in response to a clearly defined state change, the method should be invoked by an event handler. Objects that call the method should raise events instead of calling the method directly.

Some common examples of events are found in user interface applications where a user action such as clicking a button causes a segment of code to execute. The .NET event model is not limited to user interfaces. It should be used anywhere you must communicate state changes to one or more objects.

How to fix violations

If the method is called when the state of an object changes, consider changing the design to use the .NET event model.

When to suppress warnings

Suppress a warning from this rule if the method does not work with the .NET event model.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1030.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

CA1031: Do not catch general exception types

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1031
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A general exception such as System.Exception or System.SystemException is caught in a `catch` statement, or a general catch clause such as `catch()` is used.

By default, this rule only flags general exception types being caught, but this is configurable.

Rule description

General exceptions should not be caught.

How to fix violations

To fix a violation of this rule, catch a more specific exception, or rethrow the general exception as the last statement in the `catch` block.

When to suppress warnings

Do not suppress a warning from this rule. Catching general exception types can hide run-time problems from the library user and can make debugging more difficult.

> **NOTE**
>
> Starting with the .NET Framework 4, the common language runtime (CLR) no longer delivers corrupted state exceptions that occur in the operating system and managed code, such as access violations in Windows, to be handled by managed code. If you want to compile an application in the .NET Framework 4 or later versions and maintain handling of corrupted state exceptions, you can apply the HandleProcessCorruptedStateExceptionsAttribute attribute to the method that handles the corrupted state exception.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Disallowed exception type names

You can configure which exception types are disallowed from being caught. For example, to specify that the rule should flag `catch` handlers with `NullReferenceException`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA1031.disallowed_symbol_names = NullReferenceException
```

Allowed type name formats in the option value (separated by '|'):

- Type name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format with a `T:` prefix.

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA1031.disallowed_symbol_names = ExceptionType`	Matches all symbols named 'ExceptionType' in the compilation
`dotnet_code_quality.CA1031.disallowed_symbol_names = ExceptionType1|ExceptionType2`	Matches all symbols named either 'ExceptionType1' or 'ExceptionType2' in the compilation
`dotnet_code_quality.CA1031.disallowed_symbol_names = T:NS.ExceptionType`	Matches specific types named 'ExceptionType' with given fully qualified name.
`dotnet_code_quality.CA1031.disallowed_symbol_names = T:NS1.ExceptionType1|T:NS1.ExceptionType2`	Matches types named 'ExceptionType1' and 'ExceptionType2' with respective fully qualified names

You can configure these options for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows a type that violates this rule and a type that correctly implements the `catch` block.

```
using namespace System;
using namespace System::IO;

namespace DesignLibrary
{
    // Creates two violations of the rule.
    public ref class GenericExceptionsCaught
    {
        FileStream^ inStream;
        FileStream^ outStream;

    public:
        GenericExceptionsCaught(String^ inFile, String^ outFile)
        {
            try
            {
                inStream = File::Open(inFile, FileMode::Open);
            }
            catch(SystemException^ e)
            {
                Console::WriteLine("Unable to open {0}.", inFile);
            }

            try
            {
                outStream = File::Open(outFile, FileMode::Open);
            }
            catch(Exception^ e)
            {
                Console::WriteLine("Unable to open {0}.", outFile);
```

```
            }
        }
    };

    public ref class GenericExceptionsCaughtFixed
    {
        FileStream^ inStream;
        FileStream^ outStream;

    public:
        GenericExceptionsCaughtFixed(String^ inFile, String^ outFile)
        {
            try
            {
                inStream = File::Open(inFile, FileMode::Open);
            }

            // Fix the first violation by catching a specific exception.
            catch(FileNotFoundException^ e)
            {
                Console::WriteLine("Unable to open {0}.", inFile);
            }
            // For functionally equivalent code, also catch the
            // remaining exceptions that may be thrown by File.Open

            try
            {
                outStream = File::Open(outFile, FileMode::Open);
            }

            // Fix the second violation by re-throwing the generic
            // exception at the end of the catch block.
            catch(Exception^ e)
            {
                Console::WriteLine("Unable to open {0}.", outFile);
                throw;
            }
        }
    };
}
```

```
Imports System
Imports System.IO

Namespace DesignLibrary

    ' Creates two violations of the rule.
    Public Class GenericExceptionsCaught

        Dim inStream  As FileStream
        Dim outStream As FileStream

        Sub New(inFile As String, outFile As String)

            Try
                inStream = File.Open(inFile, FileMode.Open)
            Catch ex As SystemException
                Console.WriteLine("Unable to open {0}.", inFile)
            End Try

            Try
                outStream = File.Open(outFile, FileMode.Open)
            Catch
                Console.WriteLine("Unable to open {0}.", outFile)
            End Try

        End Sub

    End Class

    Public Class GenericExceptionsCaughtFixed

        Dim inStream  As FileStream
        Dim outStream As FileStream

        Sub New(inFile As String, outFile As String)

            Try
                inStream = File.Open(inFile, FileMode.Open)

            ' Fix the first violation by catching a specific exception.
            Catch ex As FileNotFoundException
                Console.WriteLine("Unable to open {0}.", inFile)
                ' For functionally equivalent code, also catch the
                ' remaining exceptions that may be thrown by File.Open
            End Try

            Try
                outStream = File.Open(outFile, FileMode.Open)

            ' Fix the second violation by re-throwing the generic
            ' exception at the end of the catch block.
            Catch
                Console.WriteLine("Unable to open {0}.", outFile)
            Throw
            End Try

        End Sub

    End Class

End Namespace
```

```
using System;
using System.IO;

namespace DesignLibrary
```

```
{
    // Creates two violations of the rule.
    public class GenericExceptionsCaught
    {
        FileStream inStream;
        FileStream outStream;

        public GenericExceptionsCaught(string inFile, string outFile)
        {
            try
            {
                inStream = File.Open(inFile, FileMode.Open);
            }
            catch(SystemException e)
            {
                Console.WriteLine("Unable to open {0}.", inFile);
            }

            try
            {
                outStream = File.Open(outFile, FileMode.Open);
            }
            catch
            {
                Console.WriteLine("Unable to open {0}.", outFile);
            }
        }
    }

    public class GenericExceptionsCaughtFixed
    {
        FileStream inStream;
        FileStream outStream;

        public GenericExceptionsCaughtFixed(string inFile, string outFile)
        {
            try
            {
                inStream = File.Open(inFile, FileMode.Open);
            }

            // Fix the first violation by catching a specific exception.
            catch(FileNotFoundException e)
            {
                Console.WriteLine("Unable to open {0}.", inFile);
            };
            // For functionally equivalent code, also catch
            // remaining exceptions that may be thrown by File.Open

            try
            {
                outStream = File.Open(outFile, FileMode.Open);
            }

            // Fix the second violation by re-throwing the generic
            // exception at the end of the catch block.
            catch
            {
                Console.WriteLine("Unable to open {0}.", outFile);
                throw;
            }
        }
    }
}
```

Related rules

CA2200: Rethrow to preserve stack details

CA1032: Implement standard exception constructors

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1032
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A type extends System.Exception but doesn't declare all the required constructors.

Rule description

Exception types must implement the following three constructors:

- public NewException()
- public NewException(string)
- public NewException(string, Exception)

Additionally, if you're running legacy FxCop analysis as opposed to .NET Compiler Platform-based FxCop analyzers, the absence of a fourth constructor also generates a violation:

- protected or private NewException(SerializationInfo, StreamingContext)

Failure to provide the full set of constructors can make it difficult to correctly handle exceptions. For example, the constructor that has the signature `NewException(string, Exception)` is used to create exceptions that are caused by other exceptions. Without this constructor, you can't create and throw an instance of your custom exception that contains an inner (nested) exception, which is what managed code should do in such a situation.

The first three exception constructors are public by convention. The fourth constructor is protected in unsealed classes, and private in sealed classes. For more information, see CA2229: Implement serialization constructors.

How to fix violations

To fix a violation of this rule, add the missing constructors to the exception, and make sure that they have the correct accessibility.

When to suppress warnings

It's safe to suppress a warning from this rule when the violation is caused by using a different access level for the public constructors. Additionally, it's okay to suppress the warning for the `NewException(SerializationInfo, StreamingContext)` constructor if you're building a Portable Class Library (PCL).

Example

The following example contains an exception type that violates this rule and an exception type that is correctly

implemented.

```
using System;
using System.Runtime.Serialization;
namespace DesignLibrary
{
   // Violates rule ImplementStandardExceptionConstructors.
   public class BadException : Exception
   {
      public BadException()
      {
         // Add any type-specific logic, and supply the default message.
      }

   }

   [Serializable()]
   public class GoodException : Exception
   {
      public GoodException()
      {
         // Add any type-specific logic, and supply the default message.
      }

      public GoodException(string message): base(message)
      {
         // Add any type-specific logic.
      }
      public GoodException(string message, Exception innerException):
         base (message, innerException)
      {
         // Add any type-specific logic for inner exceptions.
      }
      protected GoodException(SerializationInfo info,
         StreamingContext context) : base(info, context)
      {
         // Implement type-specific serialization constructor logic.
      }
   }
}
```

See also

CA2229: Implement serialization constructors

CA1033: Interface methods should be callable by child types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1033
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

An unsealed externally visible type provides an explicit method implementation of a public interface and does not provide an alternative externally visible method that has the same name.

Rule description

Consider a base type that explicitly implements a public interface method. A type that derives from the base type can access the inherited interface method only through a reference to the current instance (`this` in C#) that is cast to the interface. If the derived type reimplements (explicitly) the inherited interface method, the base implementation can no longer be accessed. The call through the current instance reference will invoke the derived implementation; this causes recursion and an eventual stack overflow.

This rule does not report a violation for an explicit implementation of System.IDisposable.Dispose when an externally visible `Close()` or `System.IDisposable.Dispose(Boolean)` method is provided.

How to fix violations

To fix a violation of this rule, implement a new method that exposes the same functionality and is visible to derived types or change to a nonexplicit implementation. If a breaking change is acceptable, an alternative is to make the type sealed.

When to suppress warnings

It is safe to suppress a warning from this rule if an externally visible method is provided that has the same functionality but a different name than the explicitly implemented method.

Example

The following example shows a type, `ViolatingBase`, that violates the rule and a type, `FixedBase`, that shows a fix for the violation.

```
using System;

namespace DesignLibrary
{
   public interface ITest
   {
      void SomeMethod();
   }

   public class ViolatingBase: ITest
   {
      void ITest.SomeMethod()
      {
         // ...
      }
   }

   public class FixedBase: ITest
   {
      void ITest.SomeMethod()
      {
         SomeMethod();
      }

      protected void SomeMethod()
      {
         // ...
      }
   }

   sealed public class Derived: FixedBase, ITest
   {
      public void SomeMethod()
      {
         // The following would cause recursion and a stack overflow.
         // ((ITest)this).SomeMethod();

         // The following is unavailable if derived from ViolatingBase.
         base.SomeMethod();
      }
   }
}
```

See also

Interfaces

CA1034: Nested types should not be visible

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1034
Category	Microsoft.Design
Breaking change	Breaking

Cause

An externally visible type contains an externally visible type declaration. Nested enumerations, protected types, and builder patterns are exempt from this rule.

Rule description

A nested type is a type declared within the scope of another type. Nested types are useful for encapsulating private implementation details of the containing type. Used for this purpose, nested types should not be externally visible.

Do not use externally visible nested types for logical grouping or to avoid name collisions; instead, use namespaces.

Nested types include the notion of member accessibility, which some programmers do not understand clearly.

Protected types can be used in subclasses and nested types in advance customization scenarios.

How to fix violations

If you do not intend the nested type to be externally visible, change the type's accessibility. Otherwise, remove the nested type from its parent. If the purpose of the nesting is to categorize the nested type, use a namespace to create the hierarchy instead.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a type that violates the rule.

```
using namespace System;

namespace DesignLibrary
{
    public ref class ParentType
    {
    public:
        ref class NestedType
        {
        public:
            NestedType()
            {
            }
        };

        ParentType()
        {
            NestedType^ nt = gcnew NestedType();
        }
    };
}
```

```
using System;

namespace DesignLibrary
{
    internal class ParentType
    {
        public class NestedType
        {
            public NestedType()
            {
            }
        }

        public ParentType()
        {
            NestedType nt = new NestedType();
        }
    }
}
```

```
Imports System

Namespace DesignLibrary

    Class ParentType

        Public Class NestedType
            Sub New()
            End Sub
        End Class

        Sub New()
        End Sub

    End Class

End Namespace
```

CA1035: ICollection implementations have strongly typed members

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1035
Category	Microsoft.Design
Breaking change	Breaking

Cause

A public or protected type implements System.Collections.ICollection but does not provide a strongly typed method for System.Collections.ICollection.CopyTo. The strongly typed version of CopyTo must accept two parameters and cannot have a System.Array or an array of System.Object as its first parameter.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule requires ICollection implementations to provide strongly typed members so that users are not required to cast arguments to the Object type when they use the functionality that is provided by the interface. This rule assumes that the type that implements ICollection does so to manage a collection of instances of a type that is stronger than Object.

ICollection implements the System.Collections.IEnumerable interface. If the objects in the collection extend System.ValueType, you must provide a strongly typed member for GetEnumerator to avoid the decrease in performance that is caused by boxing. This is not required when the objects of the collection are a reference type.

To implement a strongly typed version of an interface member, implement the interface members explicitly by using names in the form `InterfaceName.InterfaceMemberName`, such as CopyTo. The explicit interface members use the data types that are declared by the interface. Implement the strongly typed members by using the interface member name, such as CopyTo. Declare the strongly typed members as public, and declare parameters and return values to be of the strong type that is managed by the collection. The strong types replace weaker types such as Object and Array that are declared by the interface.

How to fix violations

To fix a violation of this rule, implement the interface member explicitly (declare it as CopyTo). Add the public strongly typed member, declared as `CopyTo`, and have it take a strongly typed array as its first parameter.

When to suppress warnings

Suppress a warning from this rule if you implement a new object-based collection, such as a binary tree, where

types that extend the new collection determine the strong type. These types should comply with this rule and expose strongly typed members.

Example

The following example demonstrates the correct way to implement ICollection.

```
using System;
using System.Collections;
namespace DesignLibrary
{

   public class ExceptionCollection : ICollection
   {
      private ArrayList data;

      ExceptionCollection()
      {
         data = new ArrayList();
      }

      // Provide the explicit interface member for ICollection.
      void ICollection.CopyTo(Array array, int index)
      {
         data.CopyTo(array, index);
      }

      // Provide the strongly typed member for ICollection.
      public void CopyTo(Exception[] array, int index)
      {
         ((ICollection)this).CopyTo(array, index);
      }

      // Implement the rest of the ICollection members.
      public int Count
      {
        get
        {
           return data.Count;
        }
      }

      public object SyncRoot
      {
         get
        {
           return this;
        }
      }

      public bool IsSynchronized
      {
         get
         {
            return false;
         }
      }

      // The IEnumerable interface is implemented by ICollection.
      // Because the type underlying this collection is a reference type,
      // you do not need a strongly typed version of GetEnumerator.

      public IEnumerator GetEnumerator()
      {
         return data.GetEnumerator();
      }
   }
}
```

Related rules

CA1038: Enumerators should be strongly typed

See also

- System.Array
- System.Collections.IEnumerable
- System.Collections.ICollection
- System.Object

CA1036: Override methods on comparable types

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1036
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A type implements the System.IComparable interface and does not override System.Object.Equals or does not overload the language-specific operator for equality, inequality, less-than, or greater-than. The rule does not report a violation if the type inherits only an implementation of the interface.

By default, this rule only looks at public and protected types, but this is configurable.

Rule description

Types that define a custom sort order implement the IComparable interface. The CompareTo method returns an integer value that indicates the correct sort order for two instances of the type. This rule identifies types that set a sort order. Setting a sort order implies that the ordinary meaning of equality, inequality, less-than, and greater-than don't apply. When you provide an implementation of IComparable, you must usually also override Equals so that it returns values that are consistent with CompareTo. If you override Equals and are coding in a language that supports operator overloads, you should also provide operators that are consistent with Equals.

How to fix violations

To fix a violation of this rule, override Equals. If your programming language supports operator overloading, supply the following operators:

- op_Equality
- op_Inequality
- op_LessThan
- op_GreaterThan

In C#, the tokens that are used to represent these operators are as follows:

```
==
!=
<
>
```

When to suppress warnings

It is safe to suppress a warning from rule CA1036 when the violation is caused by missing operators and your programming language does not support operator overloading, as is the case with Visual Basic. If you determine

that implementing the operators does not make sense in your app context, it's also safe to suppress a warning from this rule when it fires on equality operators other than op_Equality. However, you should always override op_Equality and the == operator if you override Object.Equals.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1036.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Examples

The following code contains a type that correctly implements IComparable. Code comments identify the methods that satisfy various rules that are related to Equals and the IComparable interface.

```
using System;
using System.Globalization;

namespace DesignLibrary
{
    // Valid ratings are between A and C.
    // A is the highest rating; it is greater than any other valid rating.
    // C is the lowest rating; it is less than any other valid rating.

    public class RatingInformation : IComparable, IComparable<RatingInformation>
    {
        public string Rating
        {
            get;
            private set;
        }

        public RatingInformation(string rating)
        {
            if (rating == null)
            {
                throw new ArgumentNullException("rating");
            }
            string v = rating.ToUpper(CultureInfo.InvariantCulture);
            if (v.Length != 1 || string.Compare(v, "C", StringComparison.Ordinal) > 0 || string.Compare(v,
"A", StringComparison.Ordinal) < 0)
            {
                throw new ArgumentException("Invalid rating value was specified.", "rating");
            }
            this.Rating = v;
        }

        public int CompareTo(object obj)
        {
            if (obj == null)
            {
                return 1;
            }
            RatingInformation other = obj as RatingInformation; // avoid double casting
            if (other == null)
            {
                throw new ArgumentException("A RatingInformation object is required for comparison.", "obj");
```

```
            }
            return this.CompareTo(other);
        }

        public int CompareTo(RatingInformation other)
        {
            if (object.ReferenceEquals(other, null))
            {
                return 1;
            }
            // Ratings compare opposite to normal string order,
            // so reverse the value returned by String.CompareTo.
            return -string.Compare(this.Rating, other.Rating, StringComparison.OrdinalIgnoreCase);
        }

        public static int Compare(RatingInformation left, RatingInformation right)
        {
            if (object.ReferenceEquals(left, right))
            {
                return 0;
            }
            if (object.ReferenceEquals(left, null))
            {
                return -1;
            }
            return left.CompareTo(right);
        }

        // Omitting Equals violates rule: OverrideMethodsOnComparableTypes.
        public override bool Equals(object obj)
        {
            RatingInformation other = obj as RatingInformation; //avoid double casting
            if (object.ReferenceEquals(other, null))
            {
                return false;
            }
            return this.CompareTo(other) == 0;
        }

        // Omitting getHashCode violates rule: OverrideGetHashCodeOnOverridingEquals.
        public override int GetHashCode()
        {
            char[] c = this.Rating.ToCharArray();
            return (int)c[0];
        }

        // Omitting any of the following operator overloads
        // violates rule: OverrideMethodsOnComparableTypes.
        public static bool operator ==(RatingInformation left, RatingInformation right)
        {
            if (object.ReferenceEquals(left, null))
            {
                return object.ReferenceEquals(right, null);
            }
            return left.Equals(right);
        }
        public static bool operator !=(RatingInformation left, RatingInformation right)
        {
            return !(left == right);
        }
        public static bool operator <(RatingInformation left, RatingInformation right)
        {
            return (Compare(left, right) < 0);
        }
        public static bool operator >(RatingInformation left, RatingInformation right)
        {
            return (Compare(left, right) > 0);
        }
    }
```

```
}
```

The following application code tests the behavior of the IComparable implementation that was shown earlier.

```
using System;

namespace DesignLibrary
{
    public class Test
    {
       public static void Main(string [] args)
       {
          if (args.Length < 2)
          {
             Console.WriteLine ("usage - TestRatings  string 1 string2");
             return;
          }
          RatingInformation r1 = new RatingInformation(args[0]) ;
          RatingInformation r2 = new RatingInformation( args[1]);
          string answer;

          if (r1.CompareTo(r2) > 0)
             answer = "greater than";
          else if (r1.CompareTo(r2) < 0)
             answer = "less than";
          else
             answer = "equal to";

          Console.WriteLine("{0} is {1} {2}", r1.Rating, answer, r2.Rating);
       }
    }
}
```

See also

- System.IComparable
- System.Object.Equals
- Equality operators

CA1038: Enumerators should be strongly typed

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1038
Category	Microsoft.Design
Breaking change	Breaking

Cause

A public or protected type implements System.Collections.IEnumerator but does not provide a strongly typed version of the System.Collections.IEnumerator.Current property. Types that are derived from the following types are exempt from this rule:

- System.Collections.CollectionBase
- System.Collections.DictionaryBase
- System.Collections.ReadOnlyCollectionBase

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule requires IEnumerator implementations to also provide a strongly typed version of the Current property so that users are not required to cast the return value to the strong type when they use the functionality that is provided by the interface. This rule assumes that the type that implements IEnumerator contains a collection of instances of a type that is stronger than Object.

How to fix violations

To fix a violation of this rule, implement the interface property explicitly (declare it as `IEnumerator.Current`). Add a public strongly typed version of the property, declared as `Current`, and have it return a strongly typed object.

When to suppress warnings

Suppress a warning from this rule when you implement an object-based enumerator for use with an object-based collection, such as a binary tree. Types that extend the new collection will define the strongly typed enumerator and expose the strongly typed property.

Example

The following example demonstrates the correct way to implement a strongly typed IEnumerator type.

```
using System;
```

```
using System;
using System.Collections;
namespace DesignLibrary
{
   // The ExceptionEnumerator class implements a strongly typed enumerator
   // for the ExceptionCollection type.

   public class ExceptionEnumerator: IEnumerator
   {
      private IEnumerator myCollectionEnumerator;

      private ExceptionEnumerator () {}

      public ExceptionEnumerator(ExceptionCollection collection)
      {
         myCollectionEnumerator = collection.data.GetEnumerator();
      }

      // Implement the IEnumerator interface member explicitly.
      object IEnumerator.Current
      {
         get
         {
            return myCollectionEnumerator.Current;
         }
      }

      // Implement the strongly typed member.
      public Exception Current
      {
         get
         {
            return (Exception) myCollectionEnumerator.Current;
         }
      }

      // Implement the remaining IEnumerator members.
      public bool MoveNext ()
      {
         return myCollectionEnumerator.MoveNext();
      }

      public void Reset ()
      {
         myCollectionEnumerator.Reset();
      }
   }

   public class ExceptionCollection : ICollection
   {
      internal ArrayList data;

      ExceptionCollection()
      {
         data = new ArrayList();
      }

      // Provide the explicit interface member for ICollection.
      void ICollection.CopyTo(Array array, int index)
      {
         data.CopyTo(array, index);
      }

      // Provide the strongly typed member for ICollection.
      public void CopyTo(Exception[] array, int index)
      {
         ((ICollection)this).CopyTo(array, index);
      }

      // Implement the rest of the ICollection members
```

```
        // Implement the rest of the ICollection members.
        public int Count
        {
          get
          {
             return data.Count;
          }
        }

        public object SyncRoot
        {
           get
          {
             return this;
          }
        }

        public bool IsSynchronized
        {
           get
           {
              return false;
           }
        }

        // The IEnumerable interface is implemented by ICollection.
        IEnumerator IEnumerable.GetEnumerator()
        {
           return new ExceptionEnumerator(this);
        }

        public ExceptionEnumerator GetEnumerator()
        {
           return new ExceptionEnumerator(this);
        }
    }
}
```

Related rules

CA1035: ICollection implementations have strongly typed members

CA1039: Lists are strongly typed

See also

- System.Collections.IEnumerator
- System.Collections.CollectionBase
- System.Collections.DictionaryBase
- System.Collections.ReadOnlyCollectionBase

CA1039: Lists are strongly typed

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1039
Category	Microsoft.Design
Breaking change	Breaking

Cause

The public or protected type implements System.Collections.IList but does not provide a strongly typed method for one or more of the following:

- IList.Item
- IList.Add
- IList.Contains
- IList.IndexOf
- IList.Insert
- IList.Remove

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule requires IList implementations to provide strongly typed members, so that users are not required to cast arguments to the System.Object type when they use the functionality that is provided by the interface. The IList interface is implemented by collections of objects that can be accessed by index. This rule assumes that the type that implements IList manages a collection of instances of a type that's stronger than Object.

IList implements the System.Collections.ICollection and System.Collections.IEnumerable interfaces. If you implement IList, you must provide the required strongly typed members for ICollection. If the objects in the collection extend System.ValueType, you must provide a strongly typed member for GetEnumerator to avoid the decrease in performance that is caused by boxing; this is not required when the objects of the collection are a reference type.

To comply with this rule, implement the interface members explicitly by using names in the form InterfaceName.InterfaceMemberName, such as Add. The explicit interface members use the data types that are declared by the interface. Implement the strongly typed members by using the interface member name, such as `Add`. Declare the strongly typed members as public, and declare parameters and return values to be of the strong type that is managed by the collection. The strong types replace weaker types such as Object and Array that are declared by the interface.

How to fix violations

To fix a violation of this rule, explicitly implement IList members and provide strongly typed alternatives for the members that were noted previously. For code that correctly implements the IList interface and provides the required strongly typed members, see the following example.

When to suppress warnings

Suppress a warning from this rule when you implement a new object-based collection, such as a linked list, where types that extend the new collection determine the strong type. These types should comply with this rule and expose strongly typed members.

Example

In the following example, the type `YourType` extends System.Collections.CollectionBase, as all strongly typed collections should. CollectionBase provides the explicit implementation of the IList interface for you. Therefore, you must only provide the strongly typed members for IList and ICollection.

```
using System;
using System.Collections;
namespace DesignLibrary
{
   public class YourType
   {
      // Implementation for your strong type goes here.

      public YourType() {}
   }

   public class YourTypeCollection : CollectionBase
   {
      // Provide the strongly typed members for IList.
      public YourType this[int index]
      {
         get
         {
            return (YourType) ((IList)this)[index];
         }
         set
         {
            ((IList)this)[index] =  value;
         }
      }

      public int Add(YourType value)
      {
         return ((IList)this).Add ((object) value);
      }

      public bool Contains(YourType value)
      {
         return ((IList)this).Contains((object) value);
      }

      public void Insert(int index, YourType value)
      {
         ((IList)this).Insert(index, (object) value);
      }

      public void Remove(YourType value)
      {
         ((IList)this).Remove((object) value);
      }

      public int IndexOf(YourType value)
      {
         return ((IList)this).IndexOf((object) value);
      }

      // Provide the strongly typed member for ICollection.

      public void CopyTo(YourType[] array, int index)
      {
         ((ICollection)this).CopyTo(array, index);
      }
   }
}
```

Related rules

CA1035: ICollection implementations have strongly typed members

CA1038: Enumerators should be strongly typed

See also

- System.Collections.CollectionBase
- System.Collections.ICollection
- System.Collections.IEnumerable
- System.Collections.IList
- System.Object

CA1040: Avoid empty interfaces

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1040
Category	Microsoft.Design
Breaking change	Breaking

Cause

The interface does not declare any members or implement two or more other interfaces.

By default, this rule only looks at externally visible interfaces, but this is configurable.

Rule description

Interfaces define members that provide a behavior or usage contract. The functionality that is described by the interface can be adopted by any type, regardless of where the type appears in the inheritance hierarchy. A type implements an interface by providing implementations for the members of the interface. An empty interface does not define any members. Therefore, it does not define a contract that can be implemented.

If your design includes empty interfaces that types are expected to implement, you are probably using an interface as a marker or a way to identify a group of types. If this identification will occur at run time, the correct way to accomplish this is to use a custom attribute. Use the presence or absence of the attribute, or the properties of the attribute, to identify the target types. If the identification must occur at compile time, then it is acceptable to use an empty interface.

How to fix violations

Remove the interface or add members to it. If the empty interface is being used to label a set of types, replace the interface with a custom attribute.

When to suppress warnings

It is safe to suppress a warning from this rule when the interface is used to identify a set of types at compile time.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1040.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows an empty interface.

```
using System;

namespace DesignLibrary
{
   public interface IBadInterface  // Violates rule
   {
   }
}
```

```
#include "stdafx.h"
using namespace System;

namespace Samples
{
    // Violates this rule
    public interface class IEmptyInterface
    {
    };
}
```

```
Imports System

Namespace Samples

    Public Interface IBadInterface  ' Violates rule
    End Interface

End Namespace
```

CA1041: Provide ObsoleteAttribute message

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1041
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A type or member is marked by using a System.ObsoleteAttribute attribute that does not have its System.ObsoleteAttribute.Message property specified.

By default, this rule only looks at externally visible types and members, but this is configurable.

Rule description

ObsoleteAttribute is used to mark deprecated library types and members. Library consumers should avoid the use of any type or member that is marked obsolete. This is because it might not be supported and will eventually be removed from later versions of the library. When a type or member marked by using ObsoleteAttribute is compiled, the Message property of the attribute is displayed. This gives the user information about the obsolete type or member. This information generally includes how long the obsolete type or member will be supported by the library designers and the preferred replacement to use.

How to fix violations

To fix a violation of this rule, add the `message` parameter to the ObsoleteAttribute constructor.

When to suppress warnings

Do not suppress a warning from this rule because the Message property provides critical information about the obsolete type or member.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1041.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows an obsolete member that has a correctly declared ObsoleteAttribute.

```
using namespace System;

namespace DesignLibrary
{
    public ref class ObsoleteAttributeOnMember
    {
    public:
        [ObsoleteAttribute("This property is obsolete and will "
            "be removed in a future version. Use the FirstName "
            "and LastName properties instead.", false)]
        property String^ Name
        {
            String^ get()
            {
               return "Name";
            }
        }

        property String^ FirstName
        {
            String^ get()
            {
               return "FirstName";
            }
        }

        property String^ LastName
        {
            String^ get()
            {
               return "LastName";
            }
        }
    };
}
```

```
using System;

namespace DesignLibrary
{
    public class ObsoleteAttributeOnMember
    {
        [ObsoleteAttribute("This property is obsolete and will " +
             "be removed in a future version. Use the FirstName " +
             "and LastName properties instead.", false)]
        public string Name
        {
            get
            {
                return "Name";
            }
        }

        public string FirstName
        {
            get
            {
                return "FirstName";
            }
        }

        public string LastName
        {
            get
            {
                return "LastName";
            }
        }

    }
}
```

```
Imports System

Namespace DesignLibrary

    Public Class ObsoleteAttributeOnMember

        <ObsoleteAttribute("This property is obsolete and will " & _
             "be removed in a future version. Use the FirstName " & _
             "and LastName properties instead.", False)> _
        ReadOnly Property Name As String
            Get
                Return "Name"
            End Get
        End Property

        ReadOnly Property FirstName As String
            Get
                Return "FirstName"
            End Get
        End Property

        ReadOnly Property LastName As String
            Get
                Return "LastName"
            End Get
        End Property

    End Class

End Namespace
```

See also

- System.ObsoleteAttribute

CA1043: Use integral or string argument for indexers

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1043
Category	Microsoft.Design
Breaking change	Breaking

Cause

A type contains an indexer that uses an index type other than System.Int32, System.Int64, System.Object, or System.String.

By default, this rule only looks at public and protected types, but this is configurable.

Rule description

Indexers, that is, indexed properties, should use integer or string types for the index. These types are typically used for indexing data structures and increase the usability of the library. Use of the Object type should be restricted to those cases where the specific integer or string type cannot be specified at design time. If the design requires other types for the index, reconsider whether the type represents a logical data store. If it does not represent a logical data store, use a method.

How to fix violations

To fix a violation of this rule, change the index to an integer or string type or use a method instead of the indexer.

When to suppress warnings

Suppress a warning from this rule only after carefully considering the need for the nonstandard indexer.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1043.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows an indexer that uses an Int32 index.

```
using System;

namespace DesignLibrary
{
    public class Months
    {
        string[] month = new string[] {"Jan", "Feb", "..."};

        public string this[int index]
        {
            get
            {
                return month[index];
            }
        }
    }
}
```

```
using namespace System;

namespace DesignLibrary
{
    public ref class Months
    {
        array<String^>^ month;

    public:
        property String^ default[int]
        {
           String^ get(int index)
           {
              return month[index];
           }
           void set(int index, String^ value)
           {
              month[index] = value;
           }
        }

        Months()
        {
            month = gcnew array<String^>(12);
            month[0] = "Jan";
            month[1] = "Feb";
            //...;
        }
    };
}
```

```
Imports System

Namespace DesignLibrary

    Public Class Months

        Private month() As String = {"Jan", "Feb", "..."}

        Default ReadOnly Property Item(index As Integer) As String
            Get
                Return month(index)
            End Get
        End Property

    End Class

End Namespace
```

Related rules

- CA1023: Indexers should not be multidimensional
- CA1024: Use properties where appropriate

CA1044: Properties should not be write only

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1044
Category	Microsoft.Design
Breaking change	Breaking

Cause

A property has a set accessor but not a get accessor.

By default, this rule only looks at public types, but this is configurable.

Rule description

Get accessors provide read access to a property and set accessors provide write access. Although it is acceptable and often necessary to have a read-only property, the design guidelines prohibit the use of write-only properties. This is because letting a user set a value and then preventing the user from viewing the value does not provide any security. Also, without read access, the state of shared objects cannot be viewed, which limits their usefulness.

How to fix violations

To fix a violation of this rule, add a get accessor to the property. Alternatively, if the behavior of a write-only property is necessary, consider converting this property to a method.

When to suppress warnings

It is recommended that you do not suppress warnings from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1044.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

In the following example, `BadClassWithWriteOnlyProperty` is a type with a write-only property. `GoodClassWithReadWriteProperty` contains the corrected code.

```
Imports System

Namespace DesignLibrary

   Public Class BadClassWithWriteOnlyProperty

      Dim someName As String

      ' Violates rule PropertiesShouldNotBeWriteOnly.
      WriteOnly Property Name As String
         Set
            someName = Value
         End Set
      End Property

   End Class

   Public Class GoodClassWithReadWriteProperty

      Dim someName As String

      Property Name As String
         Get
            Return someName
         End Get

         Set
            someName = Value
         End Set
      End Property

   End Class

End Namespace
```

```
using System;

namespace DesignLibrary
{
   public class BadClassWithWriteOnlyProperty
   {
      string someName;

      // Violates rule PropertiesShouldNotBeWriteOnly.
      public string Name
      {
         set
         {
            someName = value;
         }
      }
   }

   public class GoodClassWithReadWriteProperty
   {
      string someName;

      public string Name
      {
         get
         {
            return someName;
         }
         set
         {
            someName = value;
         }
      }
   }
}
```

CA1045: Do not pass types by reference

8/4/2020 • 6 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1045
Category	Microsoft.Design
Breaking change	Breaking

Cause

A public or protected method in a public type has a `ref` parameter that takes a primitive type, a reference type, or a value type that is not one of the built-in types.

Rule description

Passing types by reference (using `out` or `ref`) requires experience with pointers, understanding how value types and reference types differ, and handling methods that have multiple return values. Also, the difference between `out` and `ref` parameters is not widely understood.

When a reference type is passed "by reference," the method intends to use the parameter to return a different instance of the object. (Passing a reference type by reference is also known as using a double pointer, pointer to a pointer, or double indirection.) Using the default calling convention, which is pass "by value," a parameter that takes a reference type already receives a pointer to the object. The pointer, not the object to which it points, is passed by value. Passing by value means that the method cannot change the pointer to have it point to a new instance of the reference type, but can change the contents of the object to which it points. For most applications this is sufficient and yields the behavior that you want.

If a method must return a different instance, use the return value of the method to accomplish this. See the System.String class for a variety of methods that operate on strings and return a new instance of a string. By using this model, it is left to the caller to decide whether the original object is preserved.

Although return values are commonplace and heavily used, the correct application of `out` and `ref` parameters requires intermediate design and coding skills. Library architects who design for a general audience should not expect users to master working with `out` or `ref` parameters.

> **NOTE**
>
> When you work with parameters that are large structures, the additional resources that are required to copy these structures could cause a performance effect when you pass by value. In these cases, you might consider using `ref` or `out` parameters.

How to fix violations

To fix a violation of this rule that is caused by a value type, have the method return the object as its return value. If the method must return multiple values, redesign it to return a single instance of an object that holds the values.

To fix a violation of this rule that is caused by a reference type, make sure that the behavior that you want is to

return a new instance of the reference. If it is, the method should use its return value to do this.

When to suppress warnings

It is safe to suppress a warning from this rule; however, this design could cause usability issues.

Configurability

Starting with version 3.0.0 of the analyzer package, If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1045.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following library shows two implementations of a class that generates responses to the feedback of the user. The first implementation (`BadRefAndOut`) forces the library user to manage three return values. The second implementation (`RedesignedRefAndOut`) simplifies the user experience by returning an instance of a container class (`ReplyData`) that manages the data as a single unit.

```
using System;

namespace DesignLibrary
{
   public enum Actions
   {
      Unknown,
      Discard,
      ForwardToManagement,
      ForwardToDeveloper
   }

   public enum TypeOfFeedback
   {
      Complaint,
      Praise,
      Suggestion,
      Incomprehensible
   }

   public class BadRefAndOut
   {
      // Violates rule: DoNotPassTypesByReference.

      public static bool ReplyInformation (TypeOfFeedback input,
         out string reply, ref Actions action)
      {
         bool returnReply = false;
         string replyText = "Your feedback has been forwarded " +
                            "to the product manager.";

         reply = String.Empty;
         switch (input)
         {
            case TypeOfFeedback.Complaint:
            case TypeOfFeedback.Praise :
```

```
            case TypeOfFeedback.Praise :
               action = Actions.ForwardToManagement;
               reply = "Thank you. " + replyText;
               returnReply = true;
               break;
            case TypeOfFeedback.Suggestion:
               action = Actions.ForwardToDeveloper;
               reply = replyText;
               returnReply = true;
               break;
            case TypeOfFeedback.Incomprehensible:
            default:
               action = Actions.Discard;
               returnReply = false;
               break;
         }
         return returnReply;
      }
   }

   // Redesigned version does not use out or ref parameters;
   // instead, it returns this container type.

   public class ReplyData
   {
      string reply;
      Actions action;
      bool returnReply;

      // Constructors.
      public ReplyData()
      {
         this.reply = String.Empty;
         this.action = Actions.Discard;
         this.returnReply = false;
      }

      public ReplyData (Actions action, string reply, bool returnReply)
      {
         this.reply = reply;
         this.action = action;
         this.returnReply = returnReply;
      }

      // Properties.
      public string Reply { get { return reply;}}
      public Actions Action { get { return action;}}

      public override string ToString()
      {
         return String.Format("Reply: {0} Action: {1} return? {2}",
            reply, action.ToString(), returnReply.ToString());
      }
   }

   public class RedesignedRefAndOut
   {
      public static ReplyData ReplyInformation (TypeOfFeedback input)
      {
         ReplyData answer;
         string replyText = "Your feedback has been forwarded " +
            "to the product manager.";

         switch (input)
         {
            case TypeOfFeedback.Complaint:
            case TypeOfFeedback.Praise :
               answer = new ReplyData(
                  Actions.ForwardToManagement,
```

```
                "Thank you. " + replyText,
                true);
            break;
         case TypeOfFeedback.Suggestion:
            answer =  new ReplyData(
                Actions.ForwardToDeveloper,
                replyText,
                true);
            break;
         case TypeOfFeedback.Incomprehensible:
         default:
            answer = new ReplyData();
            break;
      }
      return answer;
   }
}
}
```

Example

The following application illustrates the experience of the user. The call to the redesigned library (`UseTheSimplifiedClass` method) is more straightforward, and the information that is returned by the method is easily managed. The output from the two methods is identical.

```
using System;

namespace DesignLibrary
{
   public class UseComplexMethod
   {
      static void UseTheComplicatedClass()
      {
         // Using the version with the ref and out parameters.
         // You do not have to initialize an out parameter.

         string[] reply = new string[5];

         // You must initialize a ref parameter.
         Actions[] action = {Actions.Unknown,Actions.Unknown,
                             Actions.Unknown,Actions.Unknown,
                             Actions.Unknown,Actions.Unknown};
         bool[] disposition= new bool[5];
         int i = 0;

         foreach(TypeOfFeedback t in Enum.GetValues(typeof(TypeOfFeedback)))
         {
            // The call to the library.
            disposition[i] = BadRefAndOut.ReplyInformation(
               t, out reply[i], ref action[i]);
            Console.WriteLine("Reply: {0} Action: {1}  return? {2} ",
               reply[i], action[i], disposition[i]);
            i++;
         }
      }

      static void UseTheSimplifiedClass()
      {
         ReplyData[] answer = new ReplyData[5];
         int i = 0;
         foreach(TypeOfFeedback t in Enum.GetValues(typeof(TypeOfFeedback)))
         {
            // The call to the library.
            answer[i] = RedesignedRefAndOut.ReplyInformation(t);
            Console.WriteLine(answer[i++]);
         }
      }

      public  static void Main()
      {
         UseTheComplicatedClass();

         // Print a blank line in output.
         Console.WriteLine("");

         UseTheSimplifiedClass();
      }
   }
}
```

Example

The following example library illustrates how `ref` parameters for reference types are used, and shows a better way to implement this functionality.

```
using System;

namespace DesignLibrary
{
   public class ReferenceTypesAndParameters
   {

      // The following syntax will not work. You cannot make a
      // reference type that is passed by value point to a new
      // instance. This needs the ref keyword.

      public static void BadPassTheObject(string argument)
      {
         argument = argument + " ABCDE";
      }

      // The following syntax will work, but is considered bad design.
      // It reassigns the argument to point to a new instance of string.
      // Violates rule DoNotPassTypesByReference.

      public static void PassTheReference(ref string argument)
      {
         argument = argument + " ABCDE";
      }

      // The following syntax will work and is a better design.
      // It returns the altered argument as a new instance of string.

      public static string BetterThanPassTheReference(string argument)
      {
         return argument + " ABCDE";
      }
   }
}
```

Example

The following application calls each method in the library to demonstrate the behavior.

```
using System;

namespace DesignLibrary
{
   public class Test
   {
      public static void Main()
      {
         string s1 = "12345";
         string s2 = "12345";
         string s3 = "12345";

         Console.WriteLine("Changing pointer - passed by value:");
         Console.WriteLine(s1);
         ReferenceTypesAndParameters.BadPassTheObject (s1);
         Console.WriteLine(s1);

         Console.WriteLine("Changing pointer - passed by reference:");
         Console.WriteLine(s2);
         ReferenceTypesAndParameters.PassTheReference (ref s2);
         Console.WriteLine(s2);

         Console.WriteLine("Passing by return value:");
         s3 = ReferenceTypesAndParameters.BetterThanPassTheReference (s3);
         Console.WriteLine(s3);
      }
   }
}
```

This example produces the following output:

```
Changing pointer - passed by value:
12345
12345
Changing pointer - passed by reference:
12345
12345 ABCDE
Passing by return value:
12345 ABCDE
```

Related rules

CA1021: Avoid out parameters

CA1046: Do not overload operator equals on reference types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1046
Category	Microsoft.Design
Breaking change	Breaking

Cause

A public or nested public reference type overloads the equality operator.

Rule description

For reference types, the default implementation of the equality operator is almost always correct. By default, two references are equal only if they point to the same object.

How to fix violations

To fix a violation of this rule, remove the implementation of the equality operator.

When to suppress warnings

It is safe to suppress a warning from this rule when the reference type behaves like a built-in value type. If it is meaningful to do addition or subtraction on instances of the type, it is probably correct to implement the equality operator and suppress the violation.

Example

The following example demonstrates the default behavior when comparing two references.

```
using System;

namespace DesignLibrary
{
   public class MyReferenceType
   {
      private int a, b;
      public MyReferenceType (int a, int b)
      {
         this.a = a;
         this.b = b;
      }

      public override string ToString()
      {
         return String.Format("({0},{1})", a, b);
      }
   }
}
```

Example

The following application compares some references.

```
using System;

namespace DesignLibrary
{
    public class ReferenceTypeEquality
    {
       public static void Main()
       {
          MyReferenceType a = new MyReferenceType(2,2);
          MyReferenceType b = new MyReferenceType(2,2);
          MyReferenceType c = a;

          Console.WriteLine("a = new {0} and b = new {1} are equal? {2}", a,b, a.Equals(b)? "Yes":"No");
          Console.WriteLine("c and a are equal? {0}", c.Equals(a)? "Yes":"No");
          Console.WriteLine("b and a are == ? {0}", b == a ? "Yes":"No");
          Console.WriteLine("c and a are == ? {0}", c == a ? "Yes":"No");
       }
    }
}
```

This example produces the following output:

```
a = new (2,2) and b = new (2,2) are equal? No
c and a are equal? Yes
b and a are == ? No
c and a are == ? Yes
```

Related rules

CA1013: Overload operator equals on overloading add and subtract

See also

- System.Object.Equals
- Equality Operators

CA1047: Do not declare protected members in sealed types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1047
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A public type is `sealed` (`NotInheritable` in Visual basic) and declares a protected member or a protected nested type. This rule does not report violations for Finalize methods, which must follow this pattern.

Rule description

Types declare protected members so that inheriting types can access or override the member. By definition, you cannot inherit from a sealed type, which means that protected methods on sealed types cannot be called.

The C# compiler issues a warning for this error.

How to fix violations

To fix a violation of this rule, change the access level of the member to private, or make the type inheritable.

When to suppress warnings

Do not suppress a warning from this rule. Leaving the type in its current state can cause maintenance issues and does not provide any benefits.

Configurability

Starting with version 3.0.0 of the analyzer package, If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1047.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows a type that violates this rule.

```
Imports System

Namespace DesignLibrary

   Public NotInheritable Class BadSealedType
      Protected  Sub MyMethod
      End Sub
   End Class

End Namespace
```

```
using System;

namespace DesignLibrary
{
   public sealed class SealedClass
   {
      protected void ProtectedMethod(){}
   }
}
```

CA1048: Do not declare virtual members in sealed types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1048
Category	Microsoft.Design
Breaking change	Breaking

Cause

A public type is sealed and declares a method that is both `virtual` (`Overridable` in Visual Basic) and not final. This rule does not report violations for delegate types, which must follow this pattern.

Rule description

Types declare methods as virtual so that inheriting types can override the implementation of the virtual method. By definition, you cannot inherit from a sealed type, making a virtual method on a sealed type meaningless.

The Visual Basic and C# compilers do not allow types to violate this rule.

How to fix violations

To fix a violation of this rule, make the method non-virtual or make the type inheritable.

When to suppress warnings

Do not suppress a warning from this rule. Leaving the type in its current state can cause maintenance issues and does not provide any benefits.

Example

The following example shows a type that violates this rule.

```
using namespace System;

namespace DesignLibrary
{
    public ref class SomeType sealed
    {
    public:
        virtual bool VirtualFunction() { return true; }
    };
}
```

CA1049: Types that own native resources should be disposable

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1049
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A type references a System.IntPtr field, a System.UIntPtr field, or a System.Runtime.InteropServices.HandleRef field, but does not implement System.IDisposable.

Rule description

This rule assumes that IntPtr, UIntPtr, and HandleRef fields store pointers to unmanaged resources. Types that allocate unmanaged resources should implement IDisposable to let callers to release those resources on demand and shorten the lifetimes of the objects that hold the resources.

The recommended design pattern to clean up unmanaged resources is to provide both an implicit and an explicit means to free those resources by using the System.Object.Finalize method and the System.IDisposable.Dispose method, respectively. The garbage collector calls the Finalize method of an object at some indeterminate time after the object is determined to be no longer reachable. After Finalize is called, an additional garbage collection is required to free the object. The Dispose method allows the caller to explicitly release resources on demand, earlier than the resources would be released if left to the garbage collector. After it cleans up the unmanaged resources, Dispose should call the System.GC.SuppressFinalize method to let the garbage collector know that Finalize no longer has to be called; this eliminates the need for the additional garbage collection and shortens the lifetime of the object.

How to fix violations

To fix a violation of this rule, implement IDisposable.

When to suppress warnings

It is safe to suppress a warning from this rule if the type does not reference an unmanaged resource. Otherwise, do not suppress a warning from this rule because failure to implement IDisposable can cause unmanaged resources to become unavailable or underused.

Example

The following example shows a type that implements IDisposable to clean up an unmanaged resource.

```
using System;

namespace DesignLibrary
{
    public class UnmanagedResources : IDisposable
    {
        IntPtr unmanagedResource;
        bool disposed = false;

        public UnmanagedResources()
        {
            // Allocate the unmanaged resource ...
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {
            if(!disposed)
            {
                if(disposing)
                {
                    // Release managed resources.
                }

                // Free the unmanaged resource ...

                unmanagedResource = IntPtr.Zero;

                disposed = true;
            }
        }

        ~UnmanagedResources()
        {
            Dispose(false);
        }
    }
}
```

```
Imports System

Namespace DesignLibrary

    Public Class UnmanagedResources
        Implements IDisposable

       Dim unmanagedResource As IntPtr
       Dim disposed As Boolean = False

       Sub New
           ' Allocate the unmanaged resource ...
       End Sub

       Overloads Sub Dispose() Implements IDisposable.Dispose
           Dispose(True)
           GC.SuppressFinalize(Me)
       End Sub

       Protected Overloads Overridable Sub Dispose(disposing As Boolean)
           If Not(disposed) Then

               If(disposing) Then
                   ' Release managed resources.
               End If

               ' Free the unmanaged resource ...

               unmanagedResource = IntPtr.Zero

               disposed = True

           End If
       End Sub

       Protected Overrides Sub Finalize()
           Dispose(False)
       End Sub

    End Class

End Namespace
```

Related rules

CA2115: Call GC.KeepAlive when using native resources

CA1816: Call GC.SuppressFinalize correctly

CA2216: Disposable types should declare finalizer

CA1001: Types that own disposable fields should be disposable

See also

- Cleaning Up Unmanaged Resources
- Dispose Pattern

CA1050: Declare types in namespaces

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1050
Category	Microsoft.Design
Breaking change	Breaking

Cause

A public or protected type is defined outside the scope of a named namespace.

Rule description

Types are declared in namespaces to prevent name collisions, and as a way to organize related types in an object hierarchy. Types that are outside any named namespace are in a global namespace that cannot be referenced in code.

How to fix violations

To fix a violation of this rule, place the type in a namespace.

When to suppress warnings

Although you never have to suppress a warning from this rule, it is safe to do this when the assembly will never be used together with other assemblies.

Example

The following example shows a library that has a type incorrectly declared outside a namespace, and a type that has the same name declared in a namespace.

```
using System;

// Violates rule: DeclareTypesInNamespaces.
public class Test
{
   public override string ToString()
   {
      return "Test does not live in a namespace!";
   }
}

namespace GoodSpace
{
   public class Test
   {
      public override string ToString()
      {
         return "Test lives in a namespace!";
      }
   }
}
```

```
Imports System

' Violates rule: DeclareTypesInNamespaces.
Public Class Test

    Public Overrides Function ToString() As String
        Return "Test does not live in a namespace!"
    End Function

End Class

Namespace GoodSpace

    Public Class Test

        Public Overrides Function ToString() As String
            Return "Test lives in a namespace!"
        End Function

    End Class

End Namespace
```

Example

The following application uses the library that was defined previously. Note that the type that is declared outside a namespace is created when the name `Test` is not qualified by a namespace. Note also that to access the `Test` type in `Goodspace`, the namespace name is required.

```
using System;

namespace ApplicationTester
{
    public class MainHolder
    {
        public static void Main()
        {
            Test t1 = new Test();
            Console.WriteLine(t1.ToString());

            GoodSpace.Test t2 = new GoodSpace.Test();
            Console.WriteLine(t2.ToString());
        }
    }
}
```

```
Imports System

Namespace ApplicationTester

    Public Class MainHolder

        Public Shared Sub Main()
            Dim t1 As New Test()
            Console.WriteLine(t1.ToString())

            Dim t2 As New GoodSpace.Test()
            Console.WriteLine(t2.ToString())
        End Sub

    End Class

End Namespace
```

CA1051: Do not declare visible instance fields

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1051
Category	Microsoft.Design
Breaking change	Breaking

Cause

A type has a non-private instance field.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

The primary use of a field should be as an implementation detail. Fields should be `private` or `internal` and should be exposed by using properties. It's as easy to access a property as it is to access a field, and the code in the accessors of a property can change as the features of the type expand without introducing breaking changes.

Properties that just return the value of a private or internal field are optimized to perform on par with accessing a field; the performance gain from using externally visible fields instead of properties is minimal. *Externally visible* refers to `public`, `protected`, and `protected internal` (`Public`, `Protected`, and `Protected Friend` in Visual Basic) accessibility levels.

Additionally, public fields cannot be protected by Link demands. For more information, see CA2112: Secured types should not expose fields. (Link demands are not applicable to .NET Core apps.)

How to fix violations

To fix a violation of this rule, make the field `private` or `internal` and expose it by using an externally visible property.

When to suppress warnings

Only suppress this warning if you're certain that consumers need direct access to the field. For most applications, exposed fields do not provide performance or maintainability benefits over properties.

Consumers may need field access in the following situations:

- in ASP.NET Web Forms content controls
- when the target platform makes use of `ref` to modify fields, such as model-view-viewmodel (MVVM) frameworks for WPF and UWP

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run

only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1051.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows a type (`BadPublicInstanceFields`) that violates this rule. `GoodPublicInstanceFields` shows the corrected code.

```
using System;

namespace DesignLibrary
{
   public class BadPublicInstanceFields
   {
      // Violates rule DoNotDeclareVisibleInstanceFields.
      public int instanceData = 32;
   }

   public class GoodPublicInstanceFields
   {
      private int instanceData = 32;

      public int InstanceData
      {
         get { return instanceData; }
         set { instanceData = value ; }
      }
   }
}
```

Related rules

- CA2112: Secured types should not expose fields

See also

- Link Demands

CA1052: Static holder types should be Static or NotInheritable

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1052
Category	Microsoft.Design
Breaking change	Breaking

Cause

A non-abstract type contains only static members (other than a possible default constructor) and is not declared with the static or Shared modifier.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

Rule CA1052 assumes that a type that contains only static members is not designed to be inherited, because the type does not provide any functionality that can be overridden in a derived type. A type that is not meant to be inherited should be marked with the `static` modifier in C# to prohibit its use as a base type. Additionally, its default constructor should be removed. In Visual Basic, the class should be converted to a module.

This rule does not fire for abstract classes or classes that have a base class. However, the rule does fire for classes that support an empty interface.

> **NOTE**
> In the FxCop analyzer implementation of this rule, it also encompasses the functionality of rule CA1053.

How to fix violations

To fix a violation of this rule, mark the type as `static` and remove the default constructor (C#), or convert it to a module (Visual Basic).

When to suppress warnings

You can suppress violations in the following cases:

- The type is designed to be inherited. The absence of the `static` modifier suggests that the type is useful as a base type.
- The type is used as a type argument. Static types can't be used as type arguments.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of

your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an EditorConfig file in your project:

```
dotnet_code_quality.ca1052.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example of a violation

The following example shows a type that violates the rule:

```
public class StaticMembers
{
    public static int SomeProperty { get; set; }
    public static void SomeMethod() { }
}
```

```
Imports System

Namespace DesignLibrary

    Public Class StaticMembers

        Private Shared someField As Integer

        Shared Property SomeProperty As Integer
            Get
                Return someField
            End Get
            Set
                someField = Value
            End Set
        End Property

        Private Sub New()
        End Sub

        Shared Sub SomeMethod()
        End Sub

    End Class

End Namespace
```

```
using namespace System;

namespace DesignLibrary
{
    public ref class StaticMembers
    {
        static int someField;

        StaticMembers() {}

    public:
        static property int SomeProperty
        {
            int get()
            {
                return someField;
            }

            void set(int value)
            {
                someField = value;
            }
        }

        static void SomeMethod() {}
    };
}
```

Fix with the static modifier

The following example shows how to fix a violation of this rule by marking the type with the `static` modifier in C#:

```
public static class StaticMembers
{
    public static int SomeProperty { get; set; }
    public static void SomeMethod() { }
}
```

CA1053: Static holder types should not have default constructors

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1053
Category	Microsoft.Design
Breaking change	Breaking

NOTE

Rule CA1053 is combined into CA1052: Static holder types should be sealed in FxCop analyzers.

Cause

A public or nested public type declares only static members and has a default constructor.

Rule description

The default constructor is unnecessary because calling static members does not require an instance of the type. Also, because the type does not have non-static members, creating an instance does not provide access to any of the type's members.

How to fix violations

To fix a violation of this rule, remove the default constructor.

When to suppress warnings

Do not suppress a warning from this rule. The presence of the default constructor suggests that the type is not a static type.

CA1054: URI parameters should not be strings

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1054
Category	Microsoft.Design
Breaking change	Breaking

Cause

A type declares a method with a string parameter whose name contains "uri", "Uri", "urn", "Urn", "url", or "Url" and the type does not declare a corresponding overload that takes a System.Uri parameter.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

This rule splits the parameter name into tokens based on the camel casing convention and checks whether each token equals "uri", "Uri", "urn", "Urn", "url", or "Url". If there is a match, the rule assumes that the parameter represents a uniform resource identifier (URI). A string representation of a URI is prone to parsing and encoding errors, and can lead to security vulnerabilities. If a method takes a string representation of a URI, a corresponding overload should be provided that takes an instance of the Uri class, which provides these services in a safe and secure manner.

How to fix violations

To fix a violation of this rule, change the parameter to a Uri type; this is a breaking change. Alternately, provide an overload of the method that takes a Uri parameter; this is a non-breaking change.

When to suppress warnings

It's safe to suppress a warning from this rule if the parameter does not represent a URI.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1054.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows a type, `ErrorProne`, that violates this rule, and a type, `SaferWay`, that satisfies the rule.

```
using System;

namespace DesignLibrary
{
   public class ErrorProne
   {
      string someUri;

      // Violates rule UriPropertiesShouldNotBeStrings.
      public string SomeUri
      {
         get { return someUri; }
         set { someUri = value; }
      }

      // Violates rule UriParametersShouldNotBeStrings.
      public void AddToHistory(string uriString) { }

      // Violates rule UriReturnValuesShouldNotBeStrings.
      public string GetRefererUri(string httpHeader)
      {
         return "http://www.adventure-works.com";
      }
   }

   public class SaferWay
   {
      Uri someUri;

      // To retrieve a string, call SomeUri.ToString().
      // To set using a string, call SomeUri = new Uri(string).
      public Uri SomeUri
      {
         get { return someUri; }
         set { someUri = value; }
      }

      public void AddToHistory(string uriString)
      {
         // Check for UriFormatException.
         AddToHistory(new Uri(uriString));
      }

      public void AddToHistory(Uri uriType) { }

      public Uri GetRefererUri(string httpHeader)
      {
         return new Uri("http://www.adventure-works.com");
      }
   }
}
```

```
Imports System

Namespace DesignLibrary

   Public Class ErrorProne

      Dim someUriValue As String

      ' Violates rule UriPropertiesShouldNotBeStrings.
      Property SomeUri As String
         Get
            Return someUriValue
         End Get
         Set
            someUriValue = Value
         End Set
      End Property

      ' Violates rule UriParametersShouldNotBeStrings.
      Sub AddToHistory(uriString As String)
      End Sub

      ' Violates rule UriReturnValuesShouldNotBeStrings.
      Function GetRefererUri(httpHeader As String) As String
         Return "http://www.adventure-works.com"
      End Function

   End Class

   Public Class SaferWay

      Dim someUriValue As Uri

      ' To retrieve a string, call SomeUri.ToString().
      ' To set using a string, call SomeUri = New Uri(string).
      Property SomeUri As Uri
         Get
            Return someUriValue
         End Get
         Set
            someUriValue = Value
         End Set
      End Property

      Sub AddToHistory(uriString As String)
         ' Check for UriFormatException.
         AddToHistory(New Uri(uriString))
      End Sub

      Sub AddToHistory(uriString As Uri)
      End Sub

      Function GetRefererUri(httpHeader As String) As Uri
         Return New Uri("http://www.adventure-works.com")
      End Function

   End Class

End Namespace
```

```
#using <system.dll>
using namespace System;

namespace DesignLibrary
{
   public ref class ErrorProne
   {
   public:
      // Violates rule UriPropertiesShouldNotBeStrings.
      property String^ SomeUri;

      // Violates rule UriParametersShouldNotBeStrings.
      void AddToHistory(String^ uriString) { }

      // Violates rule UriReturnValuesShouldNotBeStrings.
      String^ GetRefererUri(String^ httpHeader)
      {
         return "http://www.adventure-works.com";
      }
   };

   public ref class SaferWay
   {
   public:
      // To retrieve a string, call SomeUri()->ToString().
      // To set using a string, call SomeUri(gcnew Uri(string)).
      property Uri^ SomeUri;

      void AddToHistory(String^ uriString)
      {
         // Check for UriFormatException.
         AddToHistory(gcnew Uri(uriString));
      }

      void AddToHistory(Uri^ uriType) { }

      Uri^ GetRefererUri(String^ httpHeader)
      {
         return gcnew Uri("http://www.adventure-works.com");
      }
   };
}
```

Related rules

- CA1056: URI properties should not be strings
- CA1055: URI return values should not be strings
- CA2234: Pass System.Uri objects instead of strings
- CA1057: String URI overloads call System.Uri overloads

CA1055: URI return values should not be strings

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1055
Category	Microsoft.Design
Breaking change	Breaking

Cause

The name of a method contains "uri", "Uri", "urn", "Urn", "url", or "Url", and the method returns a string.

By default, this rule only looks at public methods, but this is configurable.

Rule description

This rule splits the method name into tokens based on the Pascal casing convention and checks whether each token equals "uri", "Uri", "urn", "Urn", "url", or "Url". If there is a match, the rule assumes that the method returns a uniform resource identifier (URI). A string representation of a URI is prone to parsing and encoding errors, and can lead to security vulnerabilities. The System.Uri class provides these services in a safe and secure manner.

How to fix violations

To fix a violation of this rule, change the return type to a Uri.

When to suppress warnings

It's safe to suppress a warning from this rule if the return value does not represent a URI.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1055.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows a type, `ErrorProne`, that violates this rule, and a type, `SaferWay`, that satisfies the rule.

```
using System;

namespace DesignLibrary
{
   public class ErrorProne
   {
      string someUri;

      // Violates rule UriPropertiesShouldNotBeStrings.
      public string SomeUri
      {
         get { return someUri; }
         set { someUri = value; }
      }

      // Violates rule UriParametersShouldNotBeStrings.
      public void AddToHistory(string uriString) { }

      // Violates rule UriReturnValuesShouldNotBeStrings.
      public string GetRefererUri(string httpHeader)
      {
         return "http://www.adventure-works.com";
      }
   }

   public class SaferWay
   {
      Uri someUri;

      // To retrieve a string, call SomeUri.ToString().
      // To set using a string, call SomeUri = new Uri(string).
      public Uri SomeUri
      {
         get { return someUri; }
         set { someUri = value; }
      }

      public void AddToHistory(string uriString)
      {
         // Check for UriFormatException.
         AddToHistory(new Uri(uriString));
      }

      public void AddToHistory(Uri uriType) { }

      public Uri GetRefererUri(string httpHeader)
      {
         return new Uri("http://www.adventure-works.com");
      }
   }
}
```

```
Imports System

Namespace DesignLibrary

   Public Class ErrorProne

      Dim someUriValue As String

      ' Violates rule UriPropertiesShouldNotBeStrings.
      Property SomeUri As String
         Get
            Return someUriValue
         End Get
         Set
            someUriValue = Value
         End Set
      End Property

      ' Violates rule UriParametersShouldNotBeStrings.
      Sub AddToHistory(uriString As String)
      End Sub

      ' Violates rule UriReturnValuesShouldNotBeStrings.
      Function GetRefererUri(httpHeader As String) As String
         Return "http://www.adventure-works.com"
      End Function

   End Class

   Public Class SaferWay

      Dim someUriValue As Uri

      ' To retrieve a string, call SomeUri.ToString().
      ' To set using a string, call SomeUri = New Uri(string).
      Property SomeUri As Uri
         Get
            Return someUriValue
         End Get
         Set
            someUriValue = Value
         End Set
      End Property

      Sub AddToHistory(uriString As String)
         ' Check for UriFormatException.
         AddToHistory(New Uri(uriString))
      End Sub

      Sub AddToHistory(uriString As Uri)
      End Sub

      Function GetRefererUri(httpHeader As String) As Uri
         Return New Uri("http://www.adventure-works.com")
      End Function

   End Class

End Namespace
```

```
#using <system.dll>
using namespace System;

namespace DesignLibrary
{
   public ref class ErrorProne
   {
   public:
      // Violates rule UriPropertiesShouldNotBeStrings.
      property String^ SomeUri;

      // Violates rule UriParametersShouldNotBeStrings.
      void AddToHistory(String^ uriString) { }

      // Violates rule UriReturnValuesShouldNotBeStrings.
      String^ GetRefererUri(String^ httpHeader)
      {
         return "http://www.adventure-works.com";
      }
   };

   public ref class SaferWay
   {
   public:
      // To retrieve a string, call SomeUri()->ToString().
      // To set using a string, call SomeUri(gcnew Uri(string)).
      property Uri^ SomeUri;

      void AddToHistory(String^ uriString)
      {
         // Check for UriFormatException.
         AddToHistory(gcnew Uri(uriString));
      }

      void AddToHistory(Uri^ uriType) { }

      Uri^ GetRefererUri(String^ httpHeader)
      {
         return gcnew Uri("http://www.adventure-works.com");
      }
   };
}
```

Related rules

- CA1056: URI properties should not be strings
- CA1054: URI parameters should not be strings
- CA2234: Pass System.Uri objects instead of strings
- CA1057: String URI overloads call System.Uri overloads

CA1056: URI properties should not be strings

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1056
Category	Microsoft.Design
Breaking change	Breaking

Cause

A type declares a string property whose name contains "uri", "Uri", "urn", "Urn", "url", or "Url".

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

This rule splits the property name into tokens based on the Pascal casing convention and checks whether each token equals "uri", "Uri", "urn", "Urn", "url", or "Url". If there is a match, the rule assumes that the property represents a uniform resource identifier (URI). A string representation of a URI is prone to parsing and encoding errors, and can lead to security vulnerabilities. The System.Uri class provides these services in a safe and secure manner.

How to fix violations

To fix a violation of this rule, change the property to a Uri type.

When to suppress warnings

It is safe to suppress a warning from this rule if the property does not represent a URI.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1056.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows a type, `ErrorProne`, that violates this rule, and a type, `SaferWay`, that satisfies the rule.

```
using System;

namespace DesignLibrary
{
   public class ErrorProne
   {
      string someUri;

      // Violates rule UriPropertiesShouldNotBeStrings.
      public string SomeUri
      {
         get { return someUri; }
         set { someUri = value; }
      }

      // Violates rule UriParametersShouldNotBeStrings.
      public void AddToHistory(string uriString) { }

      // Violates rule UriReturnValuesShouldNotBeStrings.
      public string GetRefererUri(string httpHeader)
      {
         return "http://www.adventure-works.com";
      }
   }

   public class SaferWay
   {
      Uri someUri;

      // To retrieve a string, call SomeUri.ToString().
      // To set using a string, call SomeUri = new Uri(string).
      public Uri SomeUri
      {
         get { return someUri; }
         set { someUri = value; }
      }

      public void AddToHistory(string uriString)
      {
         // Check for UriFormatException.
         AddToHistory(new Uri(uriString));
      }

      public void AddToHistory(Uri uriType) { }

      public Uri GetRefererUri(string httpHeader)
      {
         return new Uri("http://www.adventure-works.com");
      }
   }
}
```

```
Imports System

Namespace DesignLibrary

   Public Class ErrorProne

      Dim someUriValue As String

      ' Violates rule UriPropertiesShouldNotBeStrings.
      Property SomeUri As String
         Get
            Return someUriValue
         End Get
         Set
            someUriValue = Value
         End Set
      End Property

      ' Violates rule UriParametersShouldNotBeStrings.
      Sub AddToHistory(uriString As String)
      End Sub

      ' Violates rule UriReturnValuesShouldNotBeStrings.
      Function GetRefererUri(httpHeader As String) As String
         Return "http://www.adventure-works.com"
      End Function

   End Class

   Public Class SaferWay

      Dim someUriValue As Uri

      ' To retrieve a string, call SomeUri.ToString().
      ' To set using a string, call SomeUri = New Uri(string).
      Property SomeUri As Uri
         Get
            Return someUriValue
         End Get
         Set
            someUriValue = Value
         End Set
      End Property

      Sub AddToHistory(uriString As String)
         ' Check for UriFormatException.
         AddToHistory(New Uri(uriString))
      End Sub

      Sub AddToHistory(uriString As Uri)
      End Sub

      Function GetRefererUri(httpHeader As String) As Uri
         Return New Uri("http://www.adventure-works.com")
      End Function

   End Class

End Namespace
```

```
#using <system.dll>
using namespace System;

namespace DesignLibrary
{
   public ref class ErrorProne
   {
   public:
      // Violates rule UriPropertiesShouldNotBeStrings.
      property String^ SomeUri;

      // Violates rule UriParametersShouldNotBeStrings.
      void AddToHistory(String^ uriString) { }

      // Violates rule UriReturnValuesShouldNotBeStrings.
      String^ GetRefererUri(String^ httpHeader)
      {
         return "http://www.adventure-works.com";
      }
   };

   public ref class SaferWay
   {
   public:
      // To retrieve a string, call SomeUri()->ToString().
      // To set using a string, call SomeUri(gcnew Uri(string)).
      property Uri^ SomeUri;

      void AddToHistory(String^ uriString)
      {
         // Check for UriFormatException.
         AddToHistory(gcnew Uri(uriString));
      }

      void AddToHistory(Uri^ uriType) { }

      Uri^ GetRefererUri(String^ httpHeader)
      {
         return gcnew Uri("http://www.adventure-works.com");
      }
   };
}
```

Related rules

- CA1054: URI parameters should not be strings
- CA1055: URI return values should not be strings
- CA2234: Pass System.Uri objects instead of strings
- CA1057: String URI overloads call System.Uri overloads

CA1057: String URI overloads call System.Uri overloads

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1057
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A type declares method overloads that differ only by the replacement of a string parameter with a System.Uri parameter, and the overload that takes the string parameter does not call the overload that takes the Uri parameter.

Rule description

Because the overloads differ only by the string or Uri parameter, the string is assumed to represent a uniform resource identifier (URI). A string representation of a URI is prone to parsing and encoding errors, and can lead to security vulnerabilities. The Uri class provides these services in a safe and secure manner. To reap the benefits of the Uri class, the string overload should call the Uri overload using the string argument.

How to fix violations

Reimplement the method that uses the string representation of the URI so that it creates an instance of the Uri class using the string argument, and then passes the Uri object to the overload that has the Uri parameter.

When to suppress warnings

It is safe to suppress a warning from this rule if the string parameter does not represent a URI.

Example

The following example shows a correctly implemented string overload.

```
using System;

namespace DesignLibrary
{
   public class History
   {
      public void AddToHistory(string uriString)
      {
          Uri newUri = new Uri(uriString);
          AddToHistory(newUri);
      }

      public void AddToHistory(Uri uriType) { }
   }
}
```

```
#using <system.dll>
using namespace System;

namespace DesignLibrary
{
   public ref class History
   {
   public:
      void AddToHistory(String^ uriString)
      {
          Uri^ newUri = gcnew Uri(uriString);
          AddToHistory(newUri);
      }

      void AddToHistory(Uri^ uriType) { }
   };
}
```

```
Imports System

Namespace DesignLibrary

   Public Class History

      Sub AddToHistory(uriString As String)
          Dim newUri As New Uri(uriString)
          AddToHistory(newUri)
      End Sub

      Sub AddToHistory(uriType As Uri)
      End Sub

   End Class

End Namespace
```

Related rules

CA2234: Pass System.Uri objects instead of strings

CA1056: URI properties should not be strings

CA1054: URI parameters should not be strings

CA1055: URI return values should not be strings

CA1058: Types should not extend certain base types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1058
Category	Microsoft.Design
Breaking change	Breaking

Cause

A type extends one of the following base types:

- System.ApplicationException
- System.Xml.XmlDocument
- System.Collections.CollectionBase
- System.Collections.DictionaryBase
- System.Collections.Queue
- System.Collections.ReadOnlyCollectionBase
- System.Collections.SortedList
- System.Collections.Stack

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

Exceptions should derive from System.Exception or one of its subclasses in the System namespace.

Do not create a subclass of XmlDocument if you want to create an XML view of an underlying object model or data source.

Non-generic collections

Use and/or extend generic collections whenever possible. Do not extend non-generic collections in your code, unless you shipped it previously.

Examples of Incorrect Usage

```
public class MyCollection : CollectionBase
{
}

public class MyReadOnlyCollection : ReadOnlyCollectionBase
{
}
```

Examples of Correct Usage

```
public class MyCollection : Collection<T>
{
}

public class MyReadOnlyCollection : ReadOnlyCollection<T>
{
}
```

How to fix violations

To fix a violation of this rule, derive the type from a different base type or a generic collection.

When to suppress warnings

Do not suppress a warning from this rule for violations about ApplicationException. It is safe to suppress a warning from this rule for violations about XmlDocument. It is safe to suppress a warning about a non-generic collection if the code was released previously.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1058.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

CA1059: Members should not expose certain concrete types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1059
Category	Microsoft.Design
Breaking change	Breaking

Cause

An externally visible member is a certain concrete type or exposes certain concrete types through one of its parameters or return value. Currently, this rule reports exposure of the following concrete types:

- A type derived from System.Xml.XmlNode.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A concrete type is a type that has a complete implementation and therefore can be instantiated. To allow widespread use of the member, replace the concrete type with the suggested interface. This allows the member to accept any type that implements the interface or be used where a type that implements the interface is expected.

The following table lists the targeted concrete types and their suggested replacements.

CONCRETE TYPE	REPLACEMENT
XPathDocument	System.Xml.XPath.IXPathNavigable. Using the interface decouples the member from a specific implementation of an XML data source.

How to fix violations

To fix a violation of this rule, change the concrete type to the suggested interface.

When to suppress warnings

It is safe to suppress a message from this rule if the specific functionality provided by the concrete type is required.

Related rules

CA1011: Consider passing base types as parameters

CA1060: Move P/Invokes to NativeMethods class

8/4/2020 • 6 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1060
Category	Microsoft.Design
Breaking change	Breaking

Cause

A method uses Platform Invocation Services to access unmanaged code and is not a member of one of the **NativeMethods** classes.

Rule description

Platform Invocation methods, such as those that are marked by using the System.Runtime.InteropServices.DllImportAttribute attribute, or methods that are defined by using the `Declare` keyword in Visual Basic, access unmanaged code. These methods should be in one of the following classes:

- **NativeMethods** - This class does not suppress stack walks for unmanaged code permission. (System.Security.SuppressUnmanagedCodeSecurityAttribute must not be applied to this class.) This class is for methods that can be used anywhere because a stack walk will be performed.
- **SafeNativeMethods** - This class suppresses stack walks for unmanaged code permission. (System.Security.SuppressUnmanagedCodeSecurityAttribute is applied to this class.) This class is for methods that are safe for anyone to call. Callers of these methods are not required to perform a full security review to make sure that the usage is secure because the methods are harmless for any caller.
- **UnsafeNativeMethods** - This class suppresses stack walks for unmanaged code permission. (System.Security.SuppressUnmanagedCodeSecurityAttribute is applied to this class.) This class is for methods that are potentially dangerous. Any caller of these methods must perform a full security review to make sure that the usage is secure because no stack walk will be performed.

These classes are declared as `internal` (`Friend`, in Visual Basic) and declare a private constructor to prevent new instances from being created. The methods in these classes should be `static` and `internal` (`Shared` and `Friend` in Visual Basic).

How to fix violations

To fix a violation of this rule, move the method to the appropriate **NativeMethods** class. For most applications, moving P/Invokes to a new class that is named **NativeMethods** is enough.

However, if you are developing libraries for use in other applications, you should consider defining two other classes that are called **SafeNativeMethods** and **UnsafeNativeMethods**. These classes resemble the **NativeMethods** class; however, they are marked by using a special attribute called **SuppressUnmanagedCodeSecurityAttribute**. When this attribute is applied, the runtime does not perform a full stack walk to make sure that all callers have the **UnmanagedCode** permission. The runtime ordinarily

checks for this permission at startup. Because the check is not performed, it can greatly improve performance for calls to these unmanaged methods, It also enables code that has limited permissions to call these methods.

However, you should use this attribute with great care. It can have serious security implications if it is implemented incorrectly..

For information about how to implement the methods, see the **NativeMethods** Example, **SafeNativeMethods** Example, and **UnsafeNativeMethods** Example.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example declares a method that violates this rule. To correct the violation, the **RemoveDirectory** P/Invoke should be moved to an appropriate class that is designed to hold only P/Invokes.

```
Imports System

NameSpace MSInternalLibrary

' Violates rule: MovePInvokesToNativeMethodsClass.
Friend Class UnmanagedApi
    Friend Declare Function RemoveDirectory Lib "kernel32" ( _
       ByVal Name As String) As Boolean
End Class

End NameSpace
```

```
using System;
using System.Runtime.InteropServices;

namespace DesignLibrary
{
// Violates rule: MovePInvokesToNativeMethodsClass.
    internal class UnmanagedApi
    {
        [DllImport("kernel32.dll", CharSet = CharSet.Unicode)]
        internal static extern bool RemoveDirectory(string name);
    }
}
```

NativeMethods Example

Description

Because the **NativeMethods** class should not be marked by using **SuppressUnmanagedCodeSecurityAttribute**, P/Invokes that are put in it will require **UnmanagedCode** permission. Because most applications run from the local computer and run together with full trust, this is usually not a problem. However, if you are developing reusable libraries, you should consider defining a **SafeNativeMethods** or **UnsafeNativeMethods** class.

The following example shows an **Interaction.Beep** method that wraps the **MessageBeep** function from user32.dll. The **MessageBeep** P/Invoke is put in the **NativeMethods** class.

Code

```
using System;
using System.Runtime.InteropServices;
using System.ComponentModel;

public static class Interaction
{
    // Callers require Unmanaged permission
    public static void Beep()
    {
        // No need to demand a permission as callers of Interaction.Beep
        // will require UnmanagedCode permission
        if (!NativeMethods.MessageBeep(-1))
            throw new Win32Exception();
    }
}

internal static class NativeMethods
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    [return: MarshalAs(UnmanagedType.Bool)]
    internal static extern bool MessageBeep(int uType);
}
```

```
Imports System
Imports System.Runtime.InteropServices
Imports System.ComponentModel

Public NotInheritable Class Interaction

    Private Sub New()
    End Sub

    ' Callers require Unmanaged permission
    Public Shared Sub Beep()
        ' No need to demand a permission as callers of Interaction.Beep
        ' will require UnmanagedCode permission
        If Not NativeMethods.MessageBeep(-1) Then
            Throw New Win32Exception()
        End If

    End Sub

End Class

Friend NotInheritable Class NativeMethods

    Private Sub New()
    End Sub

    <DllImport("user32.dll", CharSet:=CharSet.Auto)> _
    Friend Shared Function MessageBeep(ByVal uType As Integer) As <MarshalAs(UnmanagedType.Bool)> Boolean
    End Function

End Class
```

SafeNativeMethods Example

Description

P/Invoke methods that can be safely exposed to any application and that do not have any side effects should be put in a class that is named **SafeNativeMethods**. You do not have to demand permissions and you do not have to pay much attention to where they are called from.

The following example shows an **Environment.TickCount** property that wraps the **GetTickCount** function from kernel32.dll.

Code

```
Imports System
Imports System.Runtime.InteropServices
Imports System.Security

Public NotInheritable Class Environment

    Private Sub New()
    End Sub

    ' Callers do not require Unmanaged permission
    Public Shared ReadOnly Property TickCount() As Integer
        Get
            ' No need to demand a permission in place of
            ' UnmanagedCode as GetTickCount is considered
            ' a safe method
            Return SafeNativeMethods.GetTickCount()
        End Get
    End Property

End Class

<SuppressUnmanagedCodeSecurityAttribute()> _
Friend NotInheritable Class SafeNativeMethods

    Private Sub New()
    End Sub

    <DllImport("kernel32.dll", CharSet:=CharSet.Auto, ExactSpelling:=True)> _
    Friend Shared Function GetTickCount() As Integer
    End Function

End Class
```

```
using System;
using System.Runtime.InteropServices;
using System.Security;

public static class Environment
{
    // Callers do not require UnmanagedCode permission
    public static int TickCount
    {
        get
        {
            // No need to demand a permission in place of
            // UnmanagedCode as GetTickCount is considered
            // a safe method
            return SafeNativeMethods.GetTickCount();
        }
    }
}

[SuppressUnmanagedCodeSecurityAttribute]
internal static class SafeNativeMethods
{
    [DllImport("kernel32.dll", CharSet=CharSet.Auto, ExactSpelling=true)]
    internal static extern int GetTickCount();
}
```

UnsafeNativeMethods Example

Description

P/Invoke methods that cannot be safely called and that could cause side effects should be put in a class that is named **UnsafeNativeMethods**. These methods should be rigorously checked to make sure that they are not exposed to the user unintentionally. The rule CA2118: Review SuppressUnmanagedCodeSecurityAttribute usage can help with this. Alternatively, the methods should have another permission that is demanded instead of **UnmanagedCode** when they use them.

The following example shows a **Cursor.Hide** method that wraps the **ShowCursor** function from user32.dll.

Code

```
Imports System
Imports System.Runtime.InteropServices
Imports System.Security
Imports System.Security.Permissions

Public NotInheritable Class Cursor

    Private Sub New()
    End Sub

    ' Callers do not require Unmanaged permission, however,
    ' they do require UIPermission.AllWindows
    Public Shared Sub Hide()
        ' Need to demand an appropriate permission
        ' in  place of UnmanagedCode permission as
        ' ShowCursor is not considered a safe method
        Dim permission As New UIPermission(UIPermissionWindow.AllWindows)
        permission.Demand()
        UnsafeNativeMethods.ShowCursor(False)

    End Sub

End Class

<SuppressUnmanagedCodeSecurityAttribute()> _
Friend NotInheritable Class UnsafeNativeMethods

    Private Sub New()
    End Sub

    <DllImport("user32.dll", CharSet:=CharSet.Auto, ExactSpelling:=True)> _
    Friend Shared Function ShowCursor(<MarshalAs(UnmanagedType.Bool)> ByVal bShow As Boolean) As Integer
    End Function

End Class
```

```
using System;
using System.Runtime.InteropServices;
using System.Security;
using System.Security.Permissions;

public static class Cursor
{
    // Callers do not require UnmanagedCode permission, however,
    // they do require UIPermissionWindow.AllWindows
    public static void Hide()
    {
        // Need to demand an appropriate permission
        // in  place of UnmanagedCode permission as
        // ShowCursor is not considered a safe method
        new UIPermission(UIPermissionWindow.AllWindows).Demand();
        UnsafeNativeMethods.ShowCursor(false);
    }
}

[SuppressUnmanagedCodeSecurityAttribute]
internal static class UnsafeNativeMethods
{
    [DllImport("user32.dll", CharSet = CharSet.Auto, ExactSpelling = true)]
    internal static extern int ShowCursor([MarshalAs(UnmanagedType.Bool)]bool bShow);
}
```

See also

- Design Warnings

CA1061: Do not hide base class methods

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1061
Category	Microsoft.Design
Breaking change	Breaking

Cause

A derived type declares a method with the same name and with the same number of parameters as one of its base methods; one or more of the parameters is a base type of the corresponding parameter in the base method; and any remaining parameters have types that are identical to the corresponding parameters in the base method.

Rule description

A method in a base type is hidden by an identically named method in a derived type when the parameter signature of the derived method differs only by types that are more weakly derived than the corresponding types in the parameter signature of the base method.

How to fix violations

To fix a violation of this rule, remove or rename the method, or change the parameter signature so that the method does not hide the base method.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a method that violates the rule.

```
using System;

namespace DesignLibrary
{
   class BaseType
   {
      internal void MethodOne(string inputOne, object inputTwo)
      {
         Console.WriteLine("Base: {0}, {1}", inputOne, inputTwo);
      }

      internal void MethodTwo(string inputOne, string inputTwo)
      {
         Console.WriteLine("Base: {0}, {1}", inputOne, inputTwo);
      }
   }

   class DerivedType : BaseType
   {
      internal void MethodOne(string inputOne, string inputTwo)
      {
         Console.WriteLine("Derived: {0}, {1}", inputOne, inputTwo);
      }

      // This method violates the rule.
      internal void MethodTwo(string inputOne, object inputTwo)
      {
         Console.WriteLine("Derived: {0}, {1}", inputOne, inputTwo);
      }
   }

   class Test
   {
      static void Main()
      {
         DerivedType derived = new DerivedType();

         // Calls DerivedType.MethodOne.
         derived.MethodOne("string1", "string2");

         // Calls BaseType.MethodOne.
         derived.MethodOne("string1", (object)"string2");

         // Both of these call DerivedType.MethodTwo.
         derived.MethodTwo("string1", "string2");
         derived.MethodTwo("string1", (object)"string2");
      }
   }
}
```

CA1062: Validate arguments of public methods

8/4/2020 • 5 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1062
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

An externally visible method dereferences one of its reference arguments without verifying whether that argument is `null` (`Nothing` in Visual Basic).

Following aspects of this rule are configurable:

- Whether or not to analyze extension method 'this' parameter.
- Specify null check validation methods in referenced libraries or projects, which validate that arguments passed to it are non-null.
- Parts of codebase to exclude from analysis.

Rule description

All reference arguments that are passed to externally visible methods should be checked against `null`. If appropriate, throw a ArgumentNullException when the argument is `null`.

If a method can be called from an unknown assembly because it is declared public or protected, you should validate all parameters of the method. If the method is designed to be called only by known assemblies, you should make the method internal and apply the InternalsVisibleToAttribute attribute to the assembly that contains the method.

How to fix violations

To fix a violation of this rule, validate each reference argument against `null`.

When to suppress warnings

You can suppress a warning from this rule if you are sure that the dereferenced parameter has been validated by another method call in the function.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Exclude extension method 'this' parameter

By default, this rule analyzes and flags the `this` parameter for extension methods. You can exclude analysis of the `this` parameter for extension methods by adding the following key-value pair to an .editorconfig file in your

project:

```
dotnet_code_quality.CA1062.exclude_extension_method_this_parameter = true
```

Null check validation methods

This rule can lead to false positives if your code calls special null check validation methods in referenced libraries or projects. You can avoid these false positives by specifying the name or signature of null check validation methods. The analysis will then assume that arguments passed to this method are non-null after the call. For example, to mark all methods named `Validate` as null check validation methods, you can add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA1062.null_check_validation_methods = Validate
```

Allowed method name formats in the option value (separated by '|'):

- Method name only (includes all methods with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format, with an optional `M:` prefix.

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA1062.null_check_validation_methods = Validate`	Matches all methods named 'Validate' in the compilation
`dotnet_code_quality.CA1062.null_check_validation_methods = Validate1|Validate2`	Matches all methods named either 'Validate1' or 'Validate2' in the compilation
`dotnet_code_quality.CA1062.null_check_validation_methods = NS.MyType.Validate(ParamType)`	Matches specific method 'Validate' with given fully qualified signature
`dotnet_code_quality.CA1062.null_check_validation_methods = NS1.MyType1.Validate1(ParamType)|NS2.MyType2.Validate2(ParamType)`	Matches specific methods 'Validate1' and 'Validate2' with respective fully qualified signature

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA1062.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA1062.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA1062.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA1062.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA1062.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following example shows a method that violates the rule and a method that satisfies the rule.

```
using System;

namespace DesignLibrary
{
    public class Test
    {
        // This method violates the rule.
        public void DoNotValidate(string input)
        {
            if (input.Length != 0)
            {
                Console.WriteLine(input);
            }
        }

        // This method satisfies the rule.
        public void Validate(string input)
        {
            if (input == null)
            {
                throw new ArgumentNullException(nameof(input));
            }
            if (input.Length != 0)
            {
                Console.WriteLine(input);
            }
        }
    }
}
```

```
Imports System

Namespace DesignLibrary

    Public Class Test

        ' This method violates the rule.
        Sub DoNotValidate(ByVal input As String)

            If input.Length <> 0 Then
                Console.WriteLine(input)
            End If

        End Sub

        ' This method satisfies the rule.
        Sub Validate(ByVal input As String)

            If input Is Nothing Then
                Throw New ArgumentNullException(NameOf(input))
            End If

            If input.Length <> 0 Then
                Console.WriteLine(input)
            End If

        End Sub

    End Class

End Namespace
```

Example

Copy constructors that populate field or properties that are reference objects can also violate the CA1062 rule. The violation occurs because the copied object that is passed to the copy constructor might be `null` (`Nothing` in Visual Basic). To resolve the violation, use a static (Shared in Visual Basic) method to check that the copied object is not null.

In the following `Person` class example, the `other` object that is passed to the `Person` copy constructor might be `null`.

```
public class Person
{
    public string Name { get; private set; }
    public int Age { get; private set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Copy constructor CA1062 fires because other is dereferenced
    // without being checked for null
    public Person(Person other)
        : this(other.Name, other.Age)
    {
    }
}
```

Example

In the following revised `Person` example, the `other` object that is passed to the copy constructor is first checked for null in the `PassThroughNonNull` method.

```
public class Person
{
    public string Name { get; private set; }
    public int Age { get; private set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Copy constructor
    public Person(Person other)
        : this(PassThroughNonNull(other).Name, other.Age)
    {
    }

    // Null check method
    private static Person PassThroughNonNull(Person person)
    {
        if (person == null)
            throw new ArgumentNullException(nameof(person));
        return person;
    }
}
```

CA1063: Implement IDisposable correctly

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1063
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

The System.IDisposable interface is not implemented correctly. Possible reasons for this include:

- IDisposable is reimplemented in the class.
- `Finalize` is overridden again.
- `Dispose()` is overridden.
- The `Dispose()` method is not public, sealed, or named **Dispose**.
- `Dispose(bool)` is not protected, virtual, or unsealed.
- In unsealed types, `Dispose()` must call `Dispose(true)`.
- For unsealed types, the `Finalize` implementation does not call either or both `Dispose(bool)` or the base class finalizer.

Violation of any one of these patterns triggers warning CA1063.

Every unsealed type that declares and implements the IDisposable interface must provide its own `protected virtual void Dispose(bool)` method. `Dispose()` should call `Dispose(true)`, and the finalizer should call `Dispose(false)`. If you create an unsealed type that declares and implements the IDisposable interface, you must define `Dispose(bool)` and call it. For more information, see Clean up unmanaged resources (.NET guide) and Dispose pattern.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

All IDisposable types should implement the Dispose pattern correctly.

How to fix violations

Examine your code and determine which of the following resolutions will fix this violation:

- Remove IDisposable from the list of interfaces that are implemented by your type, and override the base class Dispose implementation instead.
- Remove the finalizer from your type, override Dispose(bool disposing), and put the finalization logic in the code path where 'disposing' is false.

- Override Dispose(bool disposing), and put the dispose logic in the code path where 'disposing' is true.
- Make sure that Dispose() is declared as public and sealed.
- Rename your dispose method to **Dispose** and make sure that it's declared as public and sealed.
- Make sure that Dispose(bool) is declared as protected, virtual, and unsealed.
- Modify Dispose() so that it calls Dispose(true), then calls SuppressFinalize on the current object instance (`this` , or `Me` in Visual Basic), and then returns.
- Modify your finalizer so that it calls Dispose(false) and then returns.
- If you create an unsealed type that declares and implements the IDisposable interface, make sure that the implementation of IDisposable follows the pattern that is described earlier in this section.

When to suppress warnings

Do not suppress a warning from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1063.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Pseudo-code example

The following pseudo-code provides a general example of how Dispose(bool) should be implemented in a class that uses managed and native resources.

```
public class Resource : IDisposable
{
    private bool isDisposed;
    private IntPtr nativeResource = Marshal.AllocHGlobal(100);
    private AnotherResource managedResource = new AnotherResource();

    // Dispose() calls Dispose(true)
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // The bulk of the clean-up code is implemented in Dispose(bool)
    protected virtual void Dispose(bool disposing)
    {
        if (isDisposed) return;

        if (disposing)
        {
            // free managed resources
            managedResource.Dispose();
        }

        // free native resources if there are any.
        if (nativeResource != IntPtr.Zero)
        {
            Marshal.FreeHGlobal(nativeResource);
            nativeResource = IntPtr.Zero;
        }

        isDisposed = true;
    }

    // NOTE: Leave out the finalizer altogether if this class doesn't
    // own unmanaged resources, but leave the other methods
    // exactly as they are.
    ~Resource()
    {
        // Finalizer calls Dispose(false)
        Dispose(false);
    }
}
```

See also

- Dispose pattern (framework design guidelines)
- Clean up unmanaged resources (.NET guide)

CA1064: Exceptions should be public

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1064
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A non-public exception derives directly from Exception, SystemException, or ApplicationException.

Rule description

An internal exception is only visible inside its own internal scope. After the exception falls outside the internal scope, only the base exception can be used to catch the exception. If the internal exception is inherited from Exception, SystemException, or ApplicationException, the external code will not have sufficient information to know what to do with the exception.

But, if the code has a public exception that later is used as the base for a internal exception, it is reasonable to assume the code further out will be able to do something intelligent with the base exception. The public exception will have more information than what is provided by Exception, SystemException, or ApplicationException.

How to fix violations

Make the exception public, or derive the internal exception from a public exception that is not Exception, SystemException, or ApplicationException.

When to suppress warnings

Suppress a message from this rule if you are sure in all cases that the private exception will be caught within its own internal scope.

Example

This rule fires on the first example method, FirstCustomException because the exception class derives directly from Exception and is internal. The rule does not fire on the SecondCustomException class because although the class also derives directly from Exception, the class is declared public. The third class also does not fire the rule because it does not derive directly from System.Exception, System.SystemException, or System.ApplicationException.

```
using System;
using System.Runtime.Serialization;

namespace Samples
{
    // Violates this rule
    [Serializable]
```

```
internal class FirstCustomException : Exception
{
    internal FirstCustomException()
    {
    }

    internal FirstCustomException(string message)
        : base(message)
    {
    }

    internal FirstCustomException(string message, Exception innerException)
        : base(message, innerException)
    {
    }

    protected FirstCustomException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
    }
}

// Does not violate this rule because
// SecondCustomException is public
[Serializable]
public class SecondCustomException : Exception
{
    public SecondCustomException()
    {
    }

    public SecondCustomException(string message)
        : base(message)
    {

    }

    public SecondCustomException(string message, Exception innerException)
        : base(message, innerException)
    {
    }

    protected SecondCustomException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
    }
}

// Does not violate this rule because
// ThirdCustomException it does not derive directly from
// Exception, SystemException, or ApplicationException
[Serializable]
internal class ThirdCustomException : SecondCustomException
{
    internal ThirdCustomException()
    {
    }

    internal ThirdCustomException(string message)
        : base(message)
    {
    }

    internal ThirdCustomException(string message, Exception innerException)
        : base(message, innerException)
    {
    }
```

```
        protected ThirdCustomException(SerializationInfo info, StreamingContext context)
            : base(info, context)
        {
        }
    }
}
```

CA1065: Do not raise exceptions in unexpected locations

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1065
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A method that is not expected to throw exceptions throws an exception.

Rule description

Methods that are not expected to throw exceptions can be categorized as follows:

- Property Get Methods
- Event Accessor Methods
- Equals Methods
- GetHashCode Methods
- ToString Methods
- Static Constructors
- Finalizers
- Dispose Methods
- Equality Operators
- Implicit Cast Operators

The following sections discuss these method types.

Property Get Methods

Properties are basically smart fields. Therefore, they should behave like a field as much as possible. Fields don't throw exceptions and neither should properties. If you have a property that throws an exception, consider making it a method.

The following exceptions can be thrown from a property get method:

- System.InvalidOperationException and all derivatives (including System.ObjectDisposedException)
- System.NotSupportedException and all derivatives
- System.ArgumentException (only from indexed get)

- KeyNotFoundException (only from indexed get)

Event Accessor Methods

Event accessors should be simple operations that don't throw exceptions. An event should not throw an exception when you try to add or remove an event handler.

The following exceptions can be thrown from an event accessor:

- System.InvalidOperationException and all derivatives (including System.ObjectDisposedException)
- System.NotSupportedException and all derivatives
- ArgumentException and derivatives

Equals Methods

The following **Equals** methods should not throw exceptions:

- System.Object.Equals
- Equals

An **Equals** method should return `true` or `false` instead of throwing an exception. For example, if Equals is passed two mismatched types it should just return `false` instead of throwing an ArgumentException.

GetHashCode Methods

The following **GetHashCode** methods should usually not throw exceptions:

- GetHashCode
- GetHashCode

GetHashCode should always return a value. Otherwise, you can lose items in the hash table.

The versions of **GetHashCode** that take an argument can throw an ArgumentException. However, **Object.GetHashCode** should never throw an exception.

ToString Methods

The debugger uses System.Object.ToString to help display information about objects in string format. Therefore, **ToString** should not change the state of an object, and it shouldn't throw exceptions.

Static Constructors

Throwing exceptions from a static constructor causes the type to be unusable in the current application domain. You should have a good reason (such as a security issue) for throwing an exception from a static constructor.

Finalizers

Throwing an exception from a finalizer causes the CLR to fail fast, which tears down the process. Therefore, throwing exceptions in a finalizer should always be avoided.

Dispose Methods

A System.IDisposable.Dispose method should not throw an exception. Dispose is often called as part of the cleanup logic in a `finally` clause. Therefore, explicitly throwing an exception from Dispose forces the user to add exception handling inside the `finally` clause.

The **Dispose(false)** code path should never throw exceptions, because Dispose is almost always called from a finalizer.

Equality Operators (==, !=)

Like Equals methods, equality operators should return either `true` or `false`, and should not throw exceptions.

Implicit Cast Operators

Because the user is often unaware that an implicit cast operator has been called, an exception thrown by the implicit cast operator is unexpected. Therefore, no exceptions should be thrown from implicit cast operators.

How to fix violations

For property getters, either change the logic so that it no longer has to throw an exception, or change the property into a method.

For all other method types listed previously, change the logic so that it no longer must throw an exception.

When to suppress warnings

If the violation was caused by an exception declaration instead of a thrown exception, it is safe to suppress a warning from this rule.

Related rules

- CA2219: Do not raise exceptions in exception clauses

See also

- Design Warnings

CA1066: Implement IEquatable when overriding Equals

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1066
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A value type (struct) overrides Equals method, but does not implement IEquatable<T>.

Rule description

A value type overriding Equals method indicates that it supports comparing two instances of the type for value equality. Consider implementing the IEquatable<T> interface to support strongly typed tests for equality. This ensures that callers performing equality checks invoke the strongly typed System.IEquatable<T>.Equals method and avoid boxing the argument, improving performance. For more information, see here.

Your System.IEquatable<T>.Equals implementation should return results that are consistent with Equals.

How to fix violations

To fix a violation, implement IEquatable<T> and update Equals override to invoke this implemented method. For example, the following two code snippets show a violation of the rule and how to fix it:

```
public struct S
{
    private readonly int _value;
    public S(int f)
    {
        _value = f;
    }

    public override int GetHashCode()
        => _value.GetHashCode();

    public override bool Equals(object other)
        => other is S otherS && _value == otherS._value;
}
```

```
using System;

public struct S : IEquatable<S>
{
    private readonly int _value;
    public S(int f)
    {
        _value = f;
    }

    public override int GetHashCode()
        => _value.GetHashCode();

    public override bool Equals(object other)
        => other is S otherS && Equals(otherS);

    public bool Equals(S other)
        => _value == other._value;
}
```

When to suppress warnings

It is fine to suppress violations from this rule if the design and performance benefits from implementing the interface are not critical.

Related rules

- CA1067: Override Equals when implementing IEquatable

See also

- Design Warnings

CA1067: Override Equals when implementing IEquatable

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1067
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

A type implements IEquatable<T>, but does not override Equals method.

Rule description

A type implementing IEquatable<T> interface indicates that it supports comparing two instances of the type for equality. You should also override the base class implementations of Equals and GetHashCode() methods so that their behavior is consistent with that of the System.IEquatable<T>.Equals implementation. See here for more details.

Your Equals implementation should return results that are consistent with System.IEquatable<T>.Equals implementation.

How to fix violations

To fix a violation, override Equals and implement it by invoking the System.IEquatable<T>.Equals implementation. For example, the following two code snippets show a violation of the rule and how to fix it:

```
using System;

public struct S : IEquatable<S>
{
    private readonly int _value;
    public S(int f)
    {
        _value = f;
    }

    public bool Equals(S other)
        => _value == other._value;
}
```

```
using System;

public struct S : IEquatable<S>
{
    private readonly int _value;
    public S(int f)
    {
        _value = f;
    }

    public bool Equals(S other)
        => _value == other._value;

    public override bool Equals(object obj)
        => obj is S objS && Equals(objS);

    public override int GetHashCode()
        => _value.GetHashCode();
}
```

When to suppress warnings

Do not suppress violations of this rule.

Related rules

- CA1066: Implement IEquatable when overriding Equals

See also

- Design Warnings

CA1068: CancellationToken parameters must come last

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1068
Category	Microsoft.Design
Breaking change	Breaking

Cause

A method has a CancellationToken parameter that is not the last parameter.

Rule description

Methods that perform long running operations or asynchronous operations and are cancelable normally take a cancellation token parameter. Each cancellation token has a CancellationTokenSource that creates the token and uses it for cancelable computations. It is common practice to have a long chain of method calls that pass around the cancellation token from the callers to the callees. Hence, a large number of methods that take part in a cancelable computation end up having a cancellation token parameter. However, the cancellation token itself is not usually relevant to the core functionality of a majority of these methods. It's considered a good API design practice to have such parameters be the last parameter in the list.

Special cases

Rule CA1068 does not fire in the following special cases:

- Method has one or more optional parameters (Optional in Visual Basic) following a non-optional cancellation token parameter. The compiler requires that all optional parameters are defined after all the non-optional parameters.
- Method has one or more ref or out parameters (ByRef in Visual Basic) following a cancellation token parameter. It is common practice to have `ref` or `out` parameters be at the end of the list, because they usually indicate output values for the method.

> **NOTE**
>
> Rule CA1068 is not available in legacy analysis. It was first introduced in FxCop analyzers.

How to fix violations

Change the method signature to move the cancellation token parameter to the end of the list. For example, the following two code snippets show a violation of the rule and how to fix it:

```
// Violates CA1068
public void LongRunningOperation(CancellationToken token, string usefulParameter)
{
    ...
}
```

```
// Does not violate CA1068
public void LongRunningOperation(string usefulParameter, CancellationToken token)
{
    ...
}
```

When to suppress warnings

If the method is an externally visible public API that is already part of a shipped library, then it is safe to suppress a warning from this rule to avoid a breaking change for the library consumers.

Configurability

Roslyn analyzers issue 2851 tracks making the rule configurable in the FxCopAnalyzers package. Once it's implemented, you can configure the API surface on which rule executes to avoid having to use source suppressions.

Related rules

- CA1021: Avoid out parameters
- CA1025: Replace repetitive arguments with params array
- CA1026: Default parameters should not be used

See also

- Recommended patterns for CancellationToken

CA1069: Enums should not have duplicate values

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1069
Category	Microsoft.Design
Breaking change	Breaking

Cause

An enumeration has multiple members which are explicitly assigned the same constant value.

Rule description

Every enum member should either have a unique constant value or be explicitly assigned with a prior member in the enum to indicate explicit intent of sharing value. For example:

```
enum E
{
   Field1 = 1,
   AnotherNameForField1 = Field1,   // This is fine
   Field2 = 2,
   Field3 = 2,   // CA1069: This is not fine. Either assign a different constant value or 'Field2' to indicate
explicit intent of sharing value.
}
```

This rule helps in catching functional bugs introduced from the following scenarios:

- Accidental typing mistakes, where the user accidentally typed the same constant value for multiple members.
- Copy paste mistakes, where the user copied an existing member definition, then renamed the member but forgot to change the value.
- Merge resolution from multiple branches, where a new member was added with a different name but the same value in different branches.

> **NOTE**
>
> Rule CA1069 is not available in legacy analysis. It was first introduced in FxCop analyzers version 3.0.0.

How to fix violations

To fix a violation, either assign a new unique constant value or assign with a prior member in the enum to indicate explicit intent of sharing the same value. For example, the following code snippet shows a violation of the rule and couple of ways to fix the violation:

```
enum E
{
   Field1 = 1,
   AnotherNameForField1 = Field1,   // This is fine
   Field2 = 2,
   Field3 = 2,   // CA1069: This is not fine. Either assign a different constant value or 'Field2' to indicate
explicit intent of sharing value.
}
```

```
enum E
{
   Field1 = 1,
   AnotherNameForField1 = Field1,   // This is fine
   Field2 = 2,
   Field3 = 3,   // This is now fine
}
```

```
enum E
{
   Field1 = 1,
   AnotherNameForField1 = Field1,   // This is fine
   Field2 = 2,
   Field3 = Field2,   // This is also fine
}
```

When to suppress warnings

Do not suppress violations of this rule.

See also

- Design Warnings

CA1070: Do not declare event fields as virtual

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1070
Category	Microsoft.Design
Breaking change	Breaking

Cause

A field-like event was declared as virtual.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

Follow these .NET design guidelines to raise base class events in derived classes. Do not declare virtual events in a base class. Overridden events in a derived class have undefined behavior. The C# compiler does not handle this correctly and it is unpredictable whether a subscriber to the derived event will actually be subscribing to the base class event.

```
using System;
public class C
{
    // CA1070: Event 'ThresholdReached' should not be declared virtual.
    public virtual event EventHandler ThresholdReached;
}
```

> **NOTE**
>
> Rule CA1070 is not available in legacy analysis. It was first introduced in FxCop analyzers version 3.0.0.

How to fix violations

Follow these .NET design guidelines and avoid virtual field-like events.

When to suppress warnings

If the event is an externally visible public API that is already part of a shipped library, then it is safe to suppress a warning from this rule to avoid a breaking change for the library consumers.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1070.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

See also

- Design warnings

CA2210: Assemblies should have valid strong names

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2210
Category	Microsoft.Design
Breaking change	Non-breaking

Cause

An assembly is not signed with a strong name, the strong name could not be verified, or the strong name would not be valid without the current registry settings of the computer.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule retrieves and verifies the strong name of an assembly. A violation occurs if any of the following are true:

- The assembly does not have a strong name.
- The assembly was altered after signing.
- The assembly is delay-signed.
- The assembly was incorrectly signed, or signing failed.
- The assembly requires registry settings to pass verification. For example, the Strong Name tool (Sn.exe) was used to skip verification for the assembly.

The strong name protects clients from unknowingly loading an assembly that has been tampered with. Assemblies without strong names should not be deployed outside very limited scenarios. If you share or distribute assemblies that are not correctly signed, the assembly can be tampered with, the common language runtime might not load the assembly, or the user might have to disable verification on his or her computer. An assembly without a strong name has from the following drawbacks:

- Its origins cannot be verified.
- The common language runtime cannot warn users if the contents of the assembly have been altered.
- It cannot be loaded into the global assembly cache.

Note that to load and analyze a delay-signed assembly, you must disable verification for the assembly.

How to fix violations

Create a key file

Use one of the following procedures:

- Use the Assembly Linker tool (Al.exe).
- For the .NET Framework 2.0, use either the `/keyfile` or `/keycontainer` compiler option /KEYFILE (Specify Key or Key Pair to Sign an Assembly) or /KEYCONTAINER (Specify a Key Container to Sign an Assembly) linker option in C++).
- For the .NET Framework v1.0 or v1.1, use either the System.Reflection.AssemblyKeyFileAttribute or System.Reflection.AssemblyKeyNameAttribute attribute.

Sign your assembly with a strong name in Visual Studio

1. In Visual Studio, open your solution.
2. In **Solution Explorer**, right-click your project and then click **Properties.**
3. Click the **Signing** tab, and select the **Sign the assembly** check box.
4. From **Choose a strong name key file**, select **New**.

   The **Create Strong Name Key** window will display.
5. In **Key file name**, type a name for your strong name key.
6. Choose whether to protect the key with a password, and then click **OK**.
7. In **Solution Explorer**, right-click your project and then click **Build**.

Sign your assembly with a strong name outside Visual Studio

Use the Strong name tool (Sn.exe).

When to suppress warnings

Only suppress a warning from this rule if the assembly is used in an environment where tampering with the contents is not a concern.

See also

- System.Reflection.AssemblyKeyFileAttribute
- System.Reflection.AssemblyKeyNameAttribute
- How to: Sign an Assembly with a Strong Name
- Sn.exe (Strong Name Tool)

Documentation warnings

10/23/2019 • 2 minutes to read • Edit Online

Documentation warnings support writing well-documented libraries through the correct use of XML documentation comments for externally visible APIs.

In this section

RULE	DESCRIPTION
CA1200: Avoid using cref tags with a prefix	The cref attribute in an XML documentation tag means "code reference". It specifies that the inner text of the tag is a code element, such as a type, method, or property. Avoid using `cref` tags with prefixes, because it prevents the compiler from verifying references. It also prevents the Visual Studio integrated development environment (IDE) from finding and updating these symbol references during refactorings.

See also

- Code analysis warnings

CA1200: Avoid using cref tags with a prefix

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1200
Category	Microsoft.Documentation
Breaking change	Non-breaking

Cause

The cref tag in an XML documentation comment uses a prefix.

Rule description

The cref attribute in an XML documentation tag means "code reference". It specifies that the inner text of the tag is a code element, such as a type, method, or property. Avoid using `cref` tags with prefixes, because it prevents the compiler from verifying references. It also prevents the Visual Studio integrated development environment (IDE) from finding and updating these symbol references during refactorings. It is recommended that you use the full syntax without prefixes to reference symbol names in cref tags.

> **NOTE**
>
> Rule CA1200 is not available in legacy analysis. It was first introduced in FxCop analyzers.

How to fix violations

To fix a violation of this rule, remove the prefix from the `cref` tag. For example, the following two code snippets show a violation of the rule and how to fix it:

```
// Violates CA1200
/// <summary>
/// Type <see cref="T:C" /> contains method <see cref="F:C.F" />
/// </summary>
class C
{
    public void F() { }
}
```

```
// Does not violate CA1200
/// <summary>
/// Type <see cref="C" /> contains method <see cref="F" />
/// </summary>
class C
{
    public void F() { }
}
```

When to suppress warnings

It's safe to suppress this warning if the code reference must use a prefix because the referenced type is not findable by the compiler. For example, if a code reference references a special attribute in the full framework, but the file compiles against the portable framework, you can suppress this warning.

See also

- Documenting your code with XML comments

Globalization Warnings

1/1/2020 • 2 minutes to read • Edit Online

Globalization warnings support world-ready libraries and applications.

In This Section

RULE	DESCRIPTION
CA1300: Specify MessageBoxOptions	To correctly display a message box for cultures that use a right-to-left reading order, the RightAlign and RtlReading members of the MessageBoxOptions enumeration must be passed to the Show method.
CA1301: Avoid duplicate accelerators	An access key, also known as an accelerator, enables keyboard access to a control by using the ALT key. When multiple controls have duplicate access keys, the behavior of the access key is not well-defined.
CA1302: Do not hardcode locale specific strings	The System.Environment.SpecialFolder enumeration contains members that refer to special system folders. The locations of these folders can have different values on different operating systems; the user can change some of the locations; and the locations are localized. The Environment.GetFolderPath method returns the locations that are associated with the Environment.SpecialFolder enumeration, localized, and appropriate for the currently running computer.
CA1303: Do not pass literals as localized parameters	An externally visible method passes a string literal as a parameter to a .NET constructor or method, and that string should be localizable.
CA1304: Specify CultureInfo	A method or constructor calls a member that has an overload that accepts a System.Globalization.CultureInfo parameter, and the method or constructor does not call the overload that takes the CultureInfo parameter. When a CultureInfo or System.IFormatProvider object is not supplied, the default value that is supplied by the overloaded member might not have the effect that you want in all locales.
CA1305: Specify IFormatProvider	A method or constructor calls one or more members that have overloads that accept a System.IFormatProvider parameter, and the method or constructor does not call the overload that takes the IFormatProvider parameter. When a System.Globalization.CultureInfo or IFormatProvider object is not supplied, the default value that is supplied by the overloaded member might not have the effect that you want in all locales.
CA1306: Set locale for data types	The locale determines culture-specific presentation elements for data, such as formatting that is used for numeric values, currency symbols, and sort order. When you create a DataTable or DataSet, you should explicitly set the locale.

RULE	DESCRIPTION
CA1307: Specify StringComparison	A string comparison operation uses a method overload that does not set a StringComparison parameter.
CA1308: Normalize strings to uppercase	Strings should be normalized to uppercase. A small group of characters cannot make a round trip when they are converted to lowercase.
CA1309: Use ordinal StringComparison	A string comparison operation that is nonlinguistic does not set the StringComparison parameter to either Ordinal or OrdinalIgnoreCase. By explicitly setting the parameter to either StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase, your code often gains speed, becomes more correct, and becomes more reliable.
CA2101: Specify marshaling for P/Invoke string arguments	A platform invoke member allows for partially trusted callers, has a string parameter, and does not explicitly marshal the string. This can cause a potential security vulnerability.

CA1300: Specify MessageBoxOptions

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1300
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A method calls an overload of the System.Windows.Forms.MessageBox.Show method that does not take a System.Windows.Forms.MessageBoxOptions argument.

Rule description

To display a message box correctly for cultures that use a right-to-left reading order, pass the MessageBoxOptions.RightAlign and MessageBoxOptions.RtlReading fields to the Show method. Examine the System.Windows.Forms.Control.RightToLeft property of the containing control to determine whether to use a right-to-left reading order.

How to fix violations

To fix a violation of this rule, call an overload of the Show method that takes a MessageBoxOptions argument.

When to suppress warnings

It is safe to suppress a warning from this rule when the code library will not be localized for a culture that uses a right-to-left reading order.

Example

The following example shows a method that displays a message box that has options that are appropriate for the reading order of the culture. A resource file, which is not shown, is required to build the example. Follow the comments in the example to build the example without a resource file and to test the right-to-left feature.

```
Imports System
Imports System.Globalization
Imports System.Resources
Imports System.Windows.Forms

Namespace GlobalizationLibrary
    Class Program

        <STAThread()> _
        Shared Sub Main()
            Dim myForm As New SomeForm()

            ' Uncomment the following line to test the right to left feature.
            ' myForm.RightToLeft = RightToLeft.Yes
```

```
            Application.Run(myForm)
        End Sub
    End Class

    Public Class SomeForm : Inherits Form
        Private _Resources As ResourceManager
        Private WithEvents _Button As Button

        Public Sub New()
            _Resources = New ResourceManager(GetType(SomeForm))
            _Button = New Button()
            Controls.Add(_Button)
        End Sub

        Private Sub Button_Click(ByVal sender As Object, ByVal e As EventArgs) Handles _Button.Click
            ' Switch the commenting on the following 4 lines to test the form.
            'Dim text As String = "Text"
            'Dim caption As String = "Caption"
            Dim text As String = _Resources.GetString("messageBox.Text")
            Dim caption As String = _Resources.GetString("messageBox.Caption")

            RtlAwareMessageBox.Show(CType(sender, Control), text, caption, _
            MessageBoxButtons.OK, MessageBoxIcon.Information, _
            MessageBoxDefaultButton.Button1, CType(0, MessageBoxOptions))
        End Sub
    End Class

    Public Module RtlAwareMessageBox

        Public Function Show(ByVal owner As IWin32Window, ByVal text As String, ByVal caption As String, ByVal
buttons As MessageBoxButtons, ByVal icon As MessageBoxIcon, ByVal defaultButton As MessageBoxDefaultButton,
ByVal options As MessageBoxOptions) As DialogResult
            If (IsRightToLeft(owner)) Then
                options = options Or MessageBoxOptions.RtlReading Or _
                MessageBoxOptions.RightAlign
            End If

            Return MessageBox.Show(owner, text, caption, _
            buttons, icon, defaultButton, options)
        End Function

        Private Function IsRightToLeft(ByVal owner As IWin32Window) As Boolean
            Dim control As Control = TryCast(owner, Control)

            If (control IsNot Nothing) Then
                Return control.RightToLeft = RightToLeft.Yes
            End If

            ' If no parent control is available, ask the CurrentUICulture
            ' if we are running under right-to-left.
            Return CultureInfo.CurrentUICulture.TextInfo.IsRightToLeft
        End Function
    End Module
End Namespace
```

```
using System;
using System.Globalization;
using System.Resources;
using System.Windows.Forms;

namespace GlobalizationLibrary
{
    class Program
    {
        [STAThread]
        static void Main()
        {
```

```
            SomeForm myForm = new SomeForm();
            // Uncomment the following line to test the right-to-left feature.
            //myForm.RightToLeft = RightToLeft.Yes;
            Application.Run(myForm);
        }
    }

    public class SomeForm : Form
    {
        private ResourceManager _Resources;
        private Button _Button;
        public SomeForm()
        {
            _Resources = new ResourceManager(typeof(SomeForm));
            _Button = new Button();
            _Button.Click += new EventHandler(Button_Click);
            Controls.Add(_Button);
        }

        private void Button_Click(object sender, EventArgs e)
        {
            // Switch the commenting on the following 4 lines to test the form.
            // string text = "Text";
            // string caption = "Caption";
            string text = _Resources.GetString("messageBox.Text");
            string caption = _Resources.GetString("messageBox.Caption");
            RtlAwareMessageBox.Show((Control)sender, text, caption,
            MessageBoxButtons.OK, MessageBoxIcon.Information,
            MessageBoxDefaultButton.Button1, (MessageBoxOptions)0);
        }
    }

    public static class RtlAwareMessageBox
    {
        public static DialogResult Show(IWin32Window owner, string text, string caption, MessageBoxButtons
buttons, MessageBoxIcon icon, MessageBoxDefaultButton defaultButton, MessageBoxOptions options)
        {
            if (IsRightToLeft(owner))
            {
                options |= MessageBoxOptions.RtlReading |
                MessageBoxOptions.RightAlign;
            }

            return MessageBox.Show(owner, text, caption,
            buttons, icon, defaultButton, options);
        }

        private static bool IsRightToLeft(IWin32Window owner)
        {
            Control control = owner as Control;

            if (control != null)
            {
                return control.RightToLeft == RightToLeft.Yes;
            }

            // If no parent control is available, ask the CurrentUICulture
            // if we are running under right-to-left.
            return CultureInfo.CurrentUICulture.TextInfo.IsRightToLeft;
        }
    }
}
```

See also

- System.Resources.ResourceManager

- Resources in desktop apps (.NET Framework)

CA1301: Avoid duplicate accelerators

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1301
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A type extends System.Windows.Forms.Control and contains two or more top-level controls that have identical access keys that are stored in a resource file.

Rule description

An access key, also known as an accelerator, enables keyboard access to a control by using the **Alt** key. When multiple controls have the same access key, the behavior of the access key is not well defined. The user might not be able to access the intended control by using the access key, and a control other than the one that is intended might be enabled.

The current implementation of this rule ignores menu items. However, menu items in the same submenu should not have identical access keys.

How to fix violations

To fix a violation of this rule, define unique access keys for all controls.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a minimal form that contains two controls that have identical access keys. The keys are stored in a resource file, which is not shown. However, their values appear in the commented out `checkBox.Text` lines. The behavior of duplicate accelerators can be examined by exchanging the `checkBox.Text` lines with their commented out counterparts. However, in this case, the example will not generate a warning from the rule.

```
using System;
using System.Drawing;
using System.Resources;
using System.Windows.Forms;

namespace GlobalizationLibrary
{
   public class DuplicateAccelerators : Form
   {
      [STAThread]
      public static void Main()
      {
         DuplicateAccelerators accelerators = new DuplicateAccelerators();
         Application.Run(accelerators);
      }

      private CheckBox checkBox1;
      private CheckBox checkBox2;

      public DuplicateAccelerators()
      {
         ResourceManager resources =
            new ResourceManager(typeof(DuplicateAccelerators));

         checkBox1 = new CheckBox();
         checkBox1.Location = new Point(8, 16);
         // checkBox1.Text = "&checkBox1";
         checkBox1.Text = resources.GetString("checkBox1.Text");

         checkBox2 = new CheckBox();
         checkBox2.Location = new Point(8, 56);
         // checkBox2.Text = "&checkBox2";
         checkBox2.Text = resources.GetString("checkBox2.Text");

         Controls.Add(checkBox1);
         Controls.Add(checkBox2);
      }
   }
}
```

See also

- System.Resources.ResourceManager
- Resources in Desktop Apps

CA1302: Do not hardcode locale specific strings

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1302
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A method uses a string literal that represents part of the path of certain system folders.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

The System.Environment.SpecialFolder enumeration contains members that refer to special system folders. The locations of these folders can have different values on different operating systems, the user can change some of the locations, and the locations are localized. An example of a special folder is the System folder, which is "C:\WINDOWS\system32" on Windows XP but "C:\WINNT\system32" on Windows 2000. The System.Environment.GetFolderPath method returns the locations that are associated with the Environment.SpecialFolder enumeration. The locations that are returned by GetFolderPath are localized and appropriate for the currently running computer.

This rule tokenizes the folder paths that are retrieved by using the GetFolderPath method into separate directory levels. Each string literal is compared to the tokens. If a match is found, it is assumed that the method is building a string that refers to the system location that is associated with the token. For portability and localizability, use the GetFolderPath method to retrieve the locations of the special system folders instead of using string literals.

How to fix violations

To fix a violation of this rule, retrieve the location by using the GetFolderPath method.

When to suppress warnings

It is safe to suppress a warning from this rule if the string literal is not used to refer to one of the system locations that is associated with the Environment.SpecialFolder enumeration.

Example

The following example builds the path of the common application data folder, which generates three warnings from this rule. Next, the example retrieves the path by using the GetFolderPath method.

```
using System;

namespace GlobalizationLibrary
{
   class WriteSpecialFolders
   {
      static void Main()
      {
         string string0 = "C:";

         // Each of the following three strings violates the rule.
         string string1 = @"\Documents and Settings";
         string string2 = @"\All Users";
         string string3 = @"\Application Data";
         Console.WriteLine(string0 + string1 + string2 + string3);

         // The following statement satisfies the rule.
         Console.WriteLine(Environment.GetFolderPath(
            Environment.SpecialFolder.CommonApplicationData));
      }
   }
}
```

```
Imports System

Namespace GlobalizationLibrary

   Class WriteSpecialFolders

      Shared Sub Main()

         Dim string0 As String = "C:"

         ' Each of the following three strings violates the rule.
         Dim string1 As String = "\Documents and Settings"
         Dim string2 As String = "\All Users"
         Dim string3 As String = "\Application Data"
         Console.WriteLine(string0 & string1 & string2 & string3)

         ' The following statement satisfies the rule.
         Console.WriteLine(Environment.GetFolderPath( _
            Environment.SpecialFolder.CommonApplicationData))

      End Sub

   End Class

End Namespace
```

Related rules

CA1303: Do not pass literals as localized parameters

CA1303: Do not pass literals as localized parameters

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1303
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A method passes a string literal as a parameter to a .NET constructor or method and that string should be localizable.

This warning is raised when a literal string is passed as a value to a parameter or property and one or more of the following cases is true:

- The LocalizableAttribute attribute of the parameter or property is set to true.
- The name of the string parameter that is passed to a Console.Write or Console.WriteLine method is either "value" or "format".

By default, this rule analyzes the entire codebase, but this is configurable.

Starting with version 3.0.0 of the analyzer package, the naming heuristic is no longer used by default but this is configurable.

Rule description

String literals that are embedded in source code are difficult to localize.

How to fix violations

To fix a violation of this rule, replace the string literal with a string retrieved through an instance of the ResourceManager class.

When to suppress warnings

It is safe to suppress a warning from this rule if the code library will not be localized or if the string is not exposed to the end user or a developer using the code library.

Users can eliminate noise against methods that should not be passed localized strings by either renaming the parameter or property, or by marking these items as conditional.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded type names with derived types

You can configure which types, including its derived types, to exclude from analysis. For example, to specify that

the rule should not run on any methods within types named `MyType` and its derived types, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA1303.excluded_type_names_with_derived_types = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Type name only (includes all types with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format, with an optional `T:` prefix.

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA1303.excluded_type_names_with_derived_types = MyType`	Matches all types named 'MyType' and all of its derived types in the compilation
`dotnet_code_quality.CA1303.excluded_type_names_with_derived_types = MyType1|MyType2`	Matches all types named either 'MyType1' or 'MyType2' and all of their derived types in the compilation
`dotnet_code_quality.CA1303.excluded_type_names_with_derived_types = M:NS.MyType`	Matches specific type 'MyType' with given fully qualified name and all of its derived types
`dotnet_code_quality.CA1303.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	Matches specific types 'MyType1' and 'MyType2' with respective fully qualified names and all of their derived types

You can configure this option for just this rule, for all rules, or for all rules in this category (Globalization). For more information, see Configure FxCop analyzers.

Use naming heuristic

You can configure whether parameters or properties name containing "Text", "Message", or "Caption" will trigger this rule.

```
dotnet_code_quality.CA1303.use_naming_heuristic = true
```

Example

The following example shows a method that throws an exception when either of its two arguments are out of range. For the first argument, the exception constructor is passed a literal string, which violates this rule. For the second argument, the constructor is correctly passed a string retrieved through a ResourceManager.

```
using namespace System;
using namespace System::Globalization;
using namespace System::Reflection;
using namespace System::Resources;
using namespace System::Windows::Forms;

[assembly: NeutralResourcesLanguageAttribute("en-US")];
namespace GlobalizationLibrary
{
    public ref class DoNotPassLiterals
    {
        ResourceManager^ stringManager;

    public:
        DoNotPassLiterals()
        {
            stringManager =
                gcnew ResourceManager("en-US", Assembly::GetExecutingAssembly());
        }

        void TimeMethod(int hour, int minute)
        {
            if(hour < 0 || hour > 23)
            {
                MessageBox::Show(
                    "The valid range is 0 - 23."); //CA1303 fires because the parameter for method Show is
Text
            }

            if(minute < 0 || minute > 59)
            {
                MessageBox::Show(
                    stringManager->GetString("minuteOutOfRangeMessage", CultureInfo::CurrentUICulture));
            }
        }
    };
}
```

```
Imports System
Imports System.Globalization
Imports System.Reflection
Imports System.Resources
Imports System.Windows.Forms


<assembly: System.Resources.NeutralResourcesLanguageAttribute("en-US")>
Namespace GlobalizationLibrary

    Public Class DoNotPassLiterals

        Dim stringManager As System.Resources.ResourceManager

        Sub New()
            stringManager = New System.Resources.ResourceManager( _
                "en-US", System.Reflection.Assembly.GetExecutingAssembly())
        End Sub

        Sub TimeMethod(hour As Integer, minute As Integer)

            If(hour < 0 Or hour > 23) Then
                MessageBox.Show( _
                    "The valid range is 0 - 23.") 'CA1303 fires because the parameter for method Show is
Text
            End If

            If(minute < 0 Or minute > 59) Then
                 MessageBox.Show( _
                    stringManager.GetString("minuteOutOfRangeMessage", _
                        System.Globalization.CultureInfo.CurrentUICulture))
            End If

        End Sub

    End Class

End Namespace
```

```
using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using System.Windows.Forms;



[assembly: NeutralResourcesLanguageAttribute("en-US")]
namespace GlobalizationLibrary
{
    public class DoNotPassLiterals
    {
        ResourceManager stringManager;
        public DoNotPassLiterals()
        {
            stringManager =
            new ResourceManager("en-US", Assembly.GetExecutingAssembly());
        }

        public void TimeMethod(int hour, int minute)
        {
            if (hour < 0 || hour > 23)
            {
                MessageBox.Show(
                "The valid range is 0 - 23."); //CA1303 fires because the parameter for method Show is Text
            }

            if (minute < 0 || minute > 59)
            {
                MessageBox.Show(
                stringManager.GetString(
                "minuteOutOfRangeMessage", CultureInfo.CurrentUICulture));
            }
        }
    }

}
```

See also

- Resources in desktop apps
- Community request for change of behavior

CA1304: Specify CultureInfo

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1304
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A method or constructor calls a member that has an overload that accepts a System.Globalization.CultureInfo parameter, and the method or constructor does not call the overload that takes the CultureInfo parameter. This rule ignores calls to the following methods:

- CreateInstance
- ResourceManager.GetObject
- ResourceManager.GetString

You can also configure more symbols to be excluded by this rule.

Rule description

When a CultureInfo or System.IFormatProvider object is not supplied, the default value that is supplied by the overloaded member might not have the effect that you want in all locales. Also, .NET members choose default culture and formatting based on assumptions that might not be correct for your code. To ensure the code works as expected for your scenarios, you should supply culture-specific information according to the following guidelines:

- If the value will be displayed to the user, use the current culture. See CultureInfo.CurrentCulture.
- If the value will be stored and accessed by software, that is, persisted to a file or database, use the invariant culture. See CultureInfo.InvariantCulture.
- If you do not know the destination of the value, have the data consumer or provider specify the culture.

Even if the default behavior of the overloaded member is appropriate for your needs, it is better to explicitly call the culture-specific overload so that your code is self-documenting and more easily maintained.

> **NOTE**
>
> CultureInfo.CurrentUICulture is used only to retrieve localized resources by using an instance of the System.Resources.ResourceManager class.

How to fix violations

To fix a violation of this rule, use the overload that takes a CultureInfo argument.

When to suppress warnings

It is safe to suppress a warning from this rule when it is certain that the default culture is the correct choice, and where code maintainability is not an important development priority.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA1304.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA1304.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA1304.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA1304.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA1304.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Example showing how to fix violations

In the following example, `BadMethod` causes two violations of this rule. `GoodMethod` corrects the first violation by passing the invariant culture to String.Compare, and corrects the second violation by passing the current culture to String.ToLower because `string3` is displayed to the user.

```
using System;
using System.Globalization;

namespace GlobalizationLibrary
{
    public class CultureInfoTest
    {
        public void BadMethod(String string1, String string2, String string3)
        {
            if(string.Compare(string1, string2, false) == 0)
            {
                Console.WriteLine(string3.ToLower());
            }
        }

        public void GoodMethod(String string1, String string2, String string3)
        {
            if(string.Compare(string1, string2, false,
                              CultureInfo.InvariantCulture) == 0)
            {
                Console.WriteLine(string3.ToLower(CultureInfo.CurrentCulture));
            }
        }
    }
}
```

Example showing formatted output

The following example shows the effect of current culture on the default IFormatProvider that is selected by the DateTime type.

```
using System;
using System.Globalization;
using System.Threading;

namespace GlobalLibGlobalLibrary
{
    public class IFormatProviderTest
    {
        public static void Main()
        {
            string dt = "6/4/1900 12:15:12";

            // The default behavior of DateTime.Parse is to use
            // the current culture.

            // Violates rule: SpecifyIFormatProvider.
            DateTime myDateTime = DateTime.Parse(dt);
            Console.WriteLine(myDateTime);

            // Change the current culture to the French culture,
            // and parsing the same string yields a different value.

            Thread.CurrentThread.CurrentCulture = new CultureInfo("Fr-fr", true);
            myDateTime = DateTime.Parse(dt);

            Console.WriteLine(myDateTime);
        }
    }
}
```

This example produces the following output:

```
6/4/1900 12:15:12 PM
06/04/1900 12:15:12
```

Related rules

- CA1305: Specify IFormatProvider

See also

- Using the CultureInfo Class

CA1305: Specify IFormatProvider

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1305
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A method or constructor calls one or more members that have overloads that accept a System.IFormatProvider parameter, and the method or constructor does not call the overload that takes the IFormatProvider parameter.

This rule ignores calls to .NET methods that are documented as ignoring the IFormatProvider parameter. The rule also ignores the following methods:

- CreateInstance
- ResourceManager.GetObject
- ResourceManager.GetString
- Boolean.ToString
- Char.ToString
- Guid.ToString

Rule description

When a System.Globalization.CultureInfo or IFormatProvider object is not supplied, the default value that is supplied by the overloaded member might not have the effect that you want in all locales. Also, .NET members choose default culture and formatting based on assumptions that might not be correct for your code. To make sure that the code works as expected for your scenarios, you should supply culture-specific information according to the following guidelines:

- If the value will be displayed to the user, use the current culture. See CultureInfo.CurrentCulture.
- If the value will be stored and accessed by software (persisted to a file or database), use the invariant culture. See CultureInfo.InvariantCulture.
- If you do not know the destination of the value, have the data consumer or provider specify the culture.

Even if the default behavior of the overloaded member is appropriate for your needs, it is better to explicitly call the culture-specific overload so that your code is self-documenting and more easily maintained.

How to fix violations

To fix a violation of this rule, use the overload that takes an IFormatProvider argument. Or, use a C# interpolated string and pass it to the FormattableString.Invariant method.

When to suppress warnings

It is safe to suppress a warning from this rule when it is certain that the default format is the correct choice, and where code maintainability is not an important development priority.

Example

In the following code, the `example1` string violates rule CA1305. The `example2` string satisfies rule CA1305 by passing CultureInfo.CurrentCulture, which implements IFormatProvider, to String.Format(IFormatProvider, String, Object). The `example3` string satisfies rule CA1305 by passing an interpolated string to Invariant.

```
string name = "Georgette";

// Violates CA1305
string example1 = String.Format("Hello {0}", name);

// Satisfies CA1305
string example2 = String.Format(CultureInfo.CurrentCulture, "Hello {0}", name);

// Satisfies CA1305
string example3 = FormattableString.Invariant($"Hello {name}");
```

Related rules

- CA1304: Specify CultureInfo

See also

- Using the CultureInfo Class

CA1306: Set locale for data types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1306
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A method or constructor created one or more System.Data.DataTable or System.Data.DataSet instances and did not explicitly set the locale property (System.Data.DataTable.Locale or System.Data.DataSet.Locale).

Rule description

The locale determines culture-specific presentation elements for data, such as formatting used for numeric values, currency symbols, and sort order. When you create a DataTable or DataSet, you should set the locale explicitly. By default, the locale for these types is the current culture. For data that is stored in a database or file and is shared globally, the locale should ordinarily be set to the invariant culture (System.Globalization.CultureInfo.InvariantCulture). When data is shared across cultures, using the default locale can cause the contents of the DataTable or DataSet to be presented or interpreted incorrectly.

How to fix violations

To fix a violation of this rule, explicitly set the locale for the DataTable or DataSet.

When to suppress warnings

It is safe to suppress a warning from this rule when the library or application is for a limited local audience, the data is not shared, or the default setting yields the desired behavior in all supported scenarios.

Example

The following example creates two DataTable instances.

```
using System;
using System.Data;
using System.Globalization;

namespace GlobalLibrary
{
    public class MakeDataTables
    {
        // Violates rule: SetLocaleForDataTypes.
        public DataTable MakeBadTable()
        {
            DataTable badTable = new DataTable("Customers");
            DataColumn keyColumn = badTable.Columns.Add("ID", typeof(Int32));
            keyColumn.AllowDBNull = false;
            keyColumn.Unique = true;
            badTable.Columns.Add("LastName", typeof(String));
            badTable.Columns.Add("FirstName", typeof(String));
            return badTable;
        }

        public DataTable MakeGoodTable()
        {
            DataTable goodTable = new DataTable("Customers");
            // Satisfies rule: SetLocaleForDataTypes.
            goodTable.Locale = CultureInfo.InvariantCulture;
            DataColumn keyColumn = goodTable.Columns.Add("ID", typeof(Int32));

            keyColumn.AllowDBNull = false;
            keyColumn.Unique = true;
            goodTable.Columns.Add("LastName", typeof(String));
            goodTable.Columns.Add("FirstName", typeof(String));
            return goodTable;
        }
    }
}
```

See also

- System.Data.DataTable
- System.Data.DataSet
- System.Globalization.CultureInfo
- System.Globalization.CultureInfo.CurrentUICulture
- System.Globalization.CultureInfo.InvariantCulture

CA1307: Specify StringComparison

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1307
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A string comparison operation uses a method overload that does not set a StringComparison parameter.

Rule description

Many string operations, most important the Compare and Equals methods, provide an overload that accepts a StringComparison enumeration value as a parameter.

Whenever an overload exists that takes a StringComparison parameter, it should be used instead of an overload that does not take this parameter. By explicitly setting this parameter, your code is often made clearer and easier to maintain.

How to fix violations

To fix a violation of this rule, change string comparison methods to overloads that accept the StringComparison enumeration as a parameter. For example: change `String.Compare(str1, str2)` to `String.Compare(str1, str2, StringComparison.Ordinal)`.

When to suppress warnings

It is safe to suppress a warning from this rule when the library or application is intended for a limited local audience and will therefore not be localized.

See also

- Globalization Warnings
- CA1309: Use ordinal StringComparison

CA1308: Normalize strings to uppercase

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1308
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

An operation normalizes a string to lowercase.

Rule description

Strings should be normalized to uppercase. A small group of characters, when they are converted to lowercase, cannot make a round trip. To make a round trip means to convert the characters from one locale to another locale that represents character data differently, and then to accurately retrieve the original characters from the converted characters.

How to fix violations

Change operations that convert strings to lowercase so that the strings are converted to uppercase instead. For example, change `String.ToLower(CultureInfo.InvariantCulture)` to `String.ToUpper(CultureInfo.InvariantCulture)`.

When to suppress warnings

It is safe to suppress a warning message when you are not making security decision based on the result (for example, when you are displaying it in the UI).

See also

Globalization Warnings

CA1309: Use ordinal StringComparison

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1309
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A string comparison operation that is nonlinguistic does not set the StringComparison parameter to either **Ordinal** or **OrdinalIgnoreCase**.

Rule description

Many string operations, most importantly the System.String.Compare and System.String.Equals methods, now provide an overload that accepts a System.StringComparison enumeration value as a parameter.

When you specify either **StringComparison.Ordinal** or **StringComparison.OrdinalIgnoreCase**, the string comparison is non-linguistic. That is, the features that are specific to the natural language are ignored when comparison decisions are made. Ignoring natural language features means the decisions are based on simple byte comparisons and not on casing or equivalence tables that are parameterized by culture. As a result, by explicitly setting the parameter to either the **StringComparison.Ordinal** or **StringComparison.OrdinalIgnoreCase**, your code often gains speed, increases correctness, and becomes more reliable.

How to fix violations

To fix a violation of this rule, change the string comparison method to an overload that accepts the System.StringComparison enumeration as a parameter, and specify either **Ordinal** or **OrdinalIgnoreCase**. For example, change `String.Compare(str1, str2)` to `String.Compare(str1, str2, StringComparison.Ordinal)`.

When to suppress warnings

It is safe to suppress a warning from this rule when the library or application is intended for a limited local audience, or when the semantics of the current culture should be used.

See also

- Globalization Warnings
- CA1307: Specify StringComparison

CA2101: Specify marshaling for P/Invoke string arguments

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2101
Category	Microsoft.Globalization
Breaking change	Non-breaking

Cause

A platform invoke member allows for partially trusted callers, has a string parameter, and does not explicitly marshal the string.

Rule description

When you convert from Unicode to ANSI, it is possible that not all Unicode characters can be represented in a specific ANSI code page. *Best-fit mapping* tries to solve this problem by substituting a character for the character that cannot be represented. The use of this feature can cause a potential security vulnerability because you cannot control the character that is chosen. For example, malicious code could intentionally create a Unicode string that contains characters that are not found in a particular code page, which are converted to file system special characters such as '..' or '/'. Note also that security checks for special characters frequently occur before the string is converted to ANSI.

Best-fit mapping is the default for the unmanaged conversion, WChar to MByte. Unless you explicitly disable best-fit mapping, your code might contain an exploitable security vulnerability because of this issue.

Caution

Code Access Security (CAS) should not be considered a security boundary.

How to fix violations

To fix a violation of this rule, explicitly marshal string data types.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a method that violates this rule, and then shows how to fix the violation.

```
using System;
using System.Runtime.InteropServices;
[assembly: System.Security.AllowPartiallyTrustedCallers()]

namespace SecurityLibrary
{
    class NativeMethods
    {
        // Violates rule: SpecifyMarshalingForPInvokeStringArguments.
        [DllImport("advapi32.dll", CharSet=CharSet.Auto)]
        internal static extern int RegCreateKey(IntPtr key, String subKey, out IntPtr result);

        // Satisfies rule: SpecifyMarshalingForPInvokeStringArguments.
        [DllImport("advapi32.dll", CharSet = CharSet.Unicode)]
        internal static extern int RegCreateKey2(IntPtr key, String subKey, out IntPtr result);
    }
}
```

Interoperability Warnings

8/4/2020 • 2 minutes to read • Edit Online

Interoperability warnings support interaction with COM clients.

In This Section

RULE	DESCRIPTION
CA1400: P/Invoke entry points should exist	A public or protected method is marked by using the System.Runtime.InteropServices.DllImportAttribute attribute. Either the unmanaged library could not be located or the method could not be matched to a function in the library.
CA1401: P/Invokes should not be visible	A public or protected method in a public type has the System.Runtime.InteropServices.DllImportAttribute attribute (also implemented by the Declare keyword in Visual Basic). Such methods should not be exposed.
CA1402: Avoid overloads in COM visible interfaces	When overloaded methods are exposed to COM clients, only the first method overload retains its name. Subsequent overloads are uniquely renamed by appending to the name an underscore character (_) and an integer that corresponds to the order of declaration of the overload.
CA1403: Auto layout types should not be COM visible	A COM-visible value type is marked by using the System.Runtime.InteropServices.StructLayoutAttribute attribute set to LayoutKind.Auto. The layout of these types can change between .NET versions, which will break COM clients that expect a specific layout.
CA1404: Call GetLastError immediately after P/Invoke	A call is made to the Marshal.GetLastWin32Error method or the equivalent Win32 GetLastError function, and the immediately previous call is not to a platform invoke method.
CA1405: COM visible type base types should be COM visible	A COM-visible type derives from a type that is not COM-visible.
CA1406: Avoid Int64 arguments for Visual Basic 6 clients	Visual Basic 6 COM clients cannot access 64-bit integers.
CA1407: Avoid static members in COM visible types	COM does not support static methods.
CA1408: Do not use AutoDual ClassInterfaceType	Types that use a dual interface enable clients to bind to a specific interface layout. Any changes in a future version to the layout of the type or any base types will break COM clients that bind to the interface. By default, if the ClassInterfaceAttribute attribute is not specified, a dispatch-only interface is used.

RULE	DESCRIPTION
CA1409: Com visible types should be creatable	A reference type that is specifically marked as visible to COM contains a public parameterized constructor but does not contain a public default (parameterless) constructor. A type without a public default constructor is not creatable by COM clients.
CA1410: COM registration methods should be matched	A type declares a method that is marked by using the System.Runtime.InteropServices.ComRegisterFunctionAttribute attribute but does not declare a method that is marked by using the System.Runtime.InteropServices.ComUnregisterFunctionAttribute attribute, or vice versa.
CA1411: COM registration methods should not be visible	A method that is marked by using the System.Runtime.InteropServices.ComRegisterFunctionAttribute attribute or the System.Runtime.InteropServices.ComUnregisterFunctionAttribute attribute is externally visible.
CA1412: Mark ComSource Interfaces as IDispatch	A type is marked by using the System.Runtime.InteropServices.ComSourceInterfacesAttribute attribute, and at least one of the specified interfaces is not marked by using the System.Runtime.InteropServices.InterfaceTypeAttribute attribute set to ComInterfaceType.InterfaceIsIDispatch.
CA1413: Avoid non-public fields in COM visible value types	Nonpublic instance fields of COM-visible value types are visible to COM clients. Review the content of the fields for information that should not be exposed, or that will have unintended design or security effects.
CA1414: Mark boolean P/Invoke arguments with MarshalAs	The Boolean data type has multiple representations in unmanaged code.
CA1415: Declare P/Invokes correctly	This rule looks for platform invoke method declarations that target Win32 functions that have a pointer to an OVERLAPPED structure parameter and the corresponding managed parameter is not a pointer to a System.Threading.NativeOverlapped structure.
CA1417: Do not use `OutAttribute` on string parameters for P/Invokes	String parameters passed by value with the `OutAttribute` can destabilize the runtime if the string is an interned string.

CA1400: P/Invoke entry points should exist

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1400
Category	Microsoft.Interoperability
Breaking change	Non-breaking

Cause

A public or protected method is marked with the System.Runtime.InteropServices.DllImportAttribute. Either the unmanaged library could not be located or the method could not be matched to a function in the library. If the rule cannot find the method name exactly as it is specified, it looks for ANSI or wide-character versions of the method by suffixing the method name with 'A' or 'W'. If no match is found, the rule attempts to locate a function by using the __stdcall name format (_MyMethod@12, where 12 represents the length of the arguments). If no match is found, and the method name starts with '#', the rule searches for the function as an ordinal reference instead of a name reference.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

No compile-time check is available to make sure that methods that are marked with DllImportAttribute are located in the referenced unmanaged DLL. If no function that has the specified name is in the library, or the arguments to the method do not match the function arguments, the common language runtime throws an exception.

How to fix violations

To fix a violation of this rule, correct the method that has the DllImportAttribute attribute. Make sure that the unmanaged library exists and is in the same directory as the assembly that contains the method. If the library is present and correctly referenced, verify that the method name, return type, and argument signature match the library function.

When to suppress warnings

Do not suppress a warning from this rule when the unmanaged library is in the same directory as the managed assembly that references it. It might be safe to suppress a warning from this rule in the case where the unmanaged library could not be located.

Example

The following example shows a type that violates the rule. No function that is named `DoSomethingUnmanaged`

occurs in kernel32.dll.

```
using System.Runtime.InteropServices;

namespace InteroperabilityLibrary
{
   public class NativeMethods
   {
      // If DoSomethingUnmanaged does not exist, or has
      // a different signature or return type, the following
      // code violates rule PInvokeEntryPointsShouldExist.
      [DllImport("kernel32.dll")]
      public static extern void DoSomethingUnmanaged();
   }
}
```

See also

System.Runtime.InteropServices.DllImportAttribute

CA1401: P/Invokes should not be visible

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1401
Category	Microsoft.Interoperability
Breaking change	Breaking

Cause

A public or protected method in a public type has the System.Runtime.InteropServices.DllImportAttribute attribute (also implemented by the `Declare` keyword in Visual Basic).

Rule description

Methods that are marked with the DllImportAttribute attribute (or methods that are defined by using the `Declare` keyword in Visual Basic) use Platform Invocation Services to access unmanaged code. Such methods should not be exposed. By keeping these methods private or internal, you make sure that your library cannot be used to breach security by allowing callers access to unmanaged APIs that they could not call otherwise.

How to fix violations

To fix a violation of this rule, change the access level of the method.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example declares a method that violates this rule.

```
Imports System

NameSpace MSInternalLibrary

' Violates rule: PInvokesShouldNotBeVisible.
Public Class NativeMethods
    Public Declare Function RemoveDirectory Lib "kernel32"( _
        ByVal Name As String) As Boolean
End Class

End NameSpace
```

```
using System;
using System.Runtime.InteropServices;

namespace InteroperabilityLibrary
{
    // Violates rule: PInvokesShouldNotBeVisible.
    public class NativeMethods
    {
        [DllImport("kernel32.dll", CharSet = CharSet.Unicode)]
        public static extern bool RemoveDirectory(string name);
    }
}
```

CA1402: Avoid overloads in COM visible interfaces

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1402
Category	Microsoft.Interoperability
Breaking change	Breaking

Cause

A Component Object Model (COM) visible interface declares overloaded methods.

Rule description

When overloaded methods are exposed to COM clients, only the first method overload retains its name. Subsequent overloads are uniquely renamed by appending to the name an underscore character '_' and an integer that corresponds to the order of declaration of the overload. For example, consider the following methods:

```
void SomeMethod(int valueOne);
void SomeMethod(int valueOne, int valueTwo, int valueThree);
void SomeMethod(int valueOne, int valueTwo);
```

These methods are exposed to COM clients as the following.

```
void SomeMethod(int valueOne);
void SomeMethod_2(int valueOne, int valueTwo, int valueThree);
void SomeMethod_3(int valueOne, int valueTwo);
```

Visual Basic 6 COM clients cannot implement interface methods by using an underscore in the name.

How to fix violations

To fix a violation of this rule, rename the overloaded methods so that the names are unique. Alternatively, make the interface invisible to COM by changing the accessibility to `internal` (`Friend` in Visual Basic) or by applying the System.Runtime.InteropServices.ComVisibleAttribute attribute set to `false`.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows an interface that violates the rule and an interface that satisfies the rule.

```
Imports System
Imports System.Runtime.InteropServices

<Assembly: ComVisibleAttribute(False)>
Namespace InteroperabilityLibrary

   ' This interface violates the rule.
   <ComVisibleAttribute(True)> _
   Public Interface IOverloadedInterface

      Sub SomeSub(valueOne As Integer)
      Sub SomeSub(valueOne As Integer, valueTwo As Integer)

   End Interface

   ' This interface satisfies the rule.
   <ComVisibleAttribute(True)> _
   Public Interface INotOverloadedInterface

      Sub SomeSub(valueOne As Integer)
      Sub AnotherSub(valueOne As Integer, valueTwo As Integer)

   End Interface

End Namespace
```

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(false)]
namespace InteroperabilityLibrary
{
   // This interface violates the rule.
   [ComVisible(true)]
   public interface IOverloadedInterface
   {
      void SomeMethod(int valueOne);
      void SomeMethod(int valueOne, int valueTwo);
   }

   // This interface satisfies the rule.
   [ComVisible(true)]
   public interface INotOverloadedInterface
   {
      void SomeMethod(int valueOne);
      void AnotherMethod(int valueOne, int valueTwo);
   }
}
```

Related rules

CA1413: Avoid non-public fields in COM visible value types

CA1407: Avoid static members in COM visible types

CA1017: Mark assemblies with ComVisibleAttribute

See also

- Interoperating with Unmanaged Code
- Long Data Type

CA1403: Auto layout types should not be COM visible

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1403
Category	Microsoft.Interoperability
Breaking change	Breaking

Cause

A Component Object Model (COM) visible value type is marked with the System.Runtime.InteropServices.StructLayoutAttribute attribute set to System.Runtime.InteropServices.LayoutKind.Auto.

Rule description

LayoutKind layout types are managed by the common language runtime. The layout of these types can change between versions of .NET, which breaks COM clients that expect a specific layout. If the StructLayoutAttribute attribute is not specified, the C#, Visual Basic, and C++ compilers specify LayoutKind.Auto for value types.

Unless marked otherwise, all public, non-generic types are visible to COM, and all non-public and generic types are invisible to COM. However, to reduce false positives, this rule requires the COM visibility of the type to be explicitly stated. The containing assembly must be marked with the System.Runtime.InteropServices.ComVisibleAttribute set to `false` and the type must be marked with the ComVisibleAttribute set to `true`.

How to fix violations

To fix a violation of this rule, change the value of the StructLayoutAttribute attribute to LayoutKind.Explicit or LayoutKind.Sequential, or make the type invisible to COM.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a type that violates the rule and a type that satisfies the rule.

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(false)]
namespace InteroperabilityLibrary
{
   // This violates the rule.
   [StructLayout(LayoutKind.Auto)]
   [ComVisible(true)]
   public struct AutoLayout
   {
      public int ValueOne;
      public int ValueTwo;
   }

   // This satisfies the rule.
   [StructLayout(LayoutKind.Explicit)]
   [ComVisible(true)]
   public struct ExplicitLayout
   {
      [FieldOffset(0)]
      public int ValueOne;

      [FieldOffset(4)]
      public  int ValueTwo;
   }
}
```

```
Imports System
Imports System.Runtime.InteropServices

<Assembly: ComVisibleAttribute(False)>
Namespace InteroperabilityLibrary

   ' This violates the rule.
   <StructLayoutAttribute(LayoutKind.Auto)> _
   <ComVisibleAttribute(True)> _
   Public Structure AutoLayout

      Dim ValueOne As Integer
      Dim ValueTwo As Integer

   End Structure

   ' This satisfies the rule.
   <StructLayoutAttribute(LayoutKind.Explicit)> _
   <ComVisibleAttribute(True)> _
   Public Structure ExplicitLayout

      <FieldOffsetAttribute(0)> _
      Dim ValueOne As Integer

      <FieldOffsetAttribute(4)> _
      Dim ValueTwo As Integer

   End Structure

End Namespace
```

Related rules

CA1408: Do not use AutoDual ClassInterfaceType

See also

- Qualify .NET types for interoperation
- Interoperate with unmanaged code

CA1404: Call GetLastError immediately after P/Invoke

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1404
Category	Microsoft.Interoperability
Breaking change	Non-breaking

Cause

A call is made to the System.Runtime.InteropServices.Marshal.GetLastWin32Error method or the equivalent Win32 `GetLastError` function, and the call that comes immediately before is not to a platform invoke method.

Rule description

A platform invoke method accesses unmanaged code and is defined by using the `Declare` keyword in Visual Basic or the System.Runtime.InteropServices.DllImportAttribute attribute. Generally, upon failure, unmanaged functions call the Win32 `SetLastError` function to set an error code that is associated with the failure. The caller of the failed function calls the Win32 `GetLastError` function to retrieve the error code and determine the cause of the failure. The error code is maintained on a per-thread basis and is overwritten by the next call to `SetLastError`. After a call to a failed platform invoke method, managed code can retrieve the error code by calling the GetLastWin32Error method. Because the error code can be overwritten by internal calls from other managed class library methods, the `GetLastError` or GetLastWin32Error method should be called immediately after the platform invoke method call.

The rule ignores calls to the following managed members when they occur between the call to the platform invoke method and the call to GetLastWin32Error. These members do not change the error code and are useful for determining the success of some platform invoke method calls.

- System.IntPtr.Zero
- System.IntPtr.Equality
- System.IntPtr.Inequality
- System.Runtime.InteropServices.SafeHandle.IsInvalid

How to fix violations

To fix a violation of this rule, move the call to GetLastWin32Error so that it immediately follows the call to the platform invoke method.

When to suppress warnings

It is safe to suppress a warning from this rule if the code between the platform invoke method call and the GetLastWin32Error method call cannot explicitly or implicitly cause the error code to change.

Example

The following example shows a method that violates the rule and a method that satisfies the rule.

```
Imports System
Imports System.Runtime.InteropServices
Imports System.Text

Namespace InteroperabilityLibrary

   Class NativeMethods

      Private Sub New()
      End Sub

      ' Violates rule UseManagedEquivalentsOfWin32Api.
      Friend Declare Auto Function _
         ExpandEnvironmentStrings Lib "kernel32.dll" _
         (lpSrc As String, lpDst As StringBuilder, nSize As Integer) _
         As Integer

   End Class

   Public Class UseNativeMethod

      Dim environmentVariable As String = "%TEMP%"
      Dim expandedVariable As StringBuilder

      Sub ViolateRule()

         expandedVariable = New StringBuilder(100)

         If NativeMethods.ExpandEnvironmentStrings( _
            environmentVariable, _
            expandedVariable, _
            expandedVariable.Capacity) = 0

            ' Violates rule CallGetLastErrorImmediatelyAfterPInvoke.
            Console.Error.WriteLine(Marshal.GetLastWin32Error())
         Else
            Console.WriteLine(expandedVariable)
         End If

      End Sub

      Sub SatisfyRule()

         expandedVariable = New StringBuilder(100)

         If NativeMethods.ExpandEnvironmentStrings( _
            environmentVariable, _
            expandedVariable, _
            expandedVariable.Capacity) = 0

            ' Satisfies rule CallGetLastErrorImmediatelyAfterPInvoke.
            Dim lastError As Integer = Marshal.GetLastWin32Error()
            Console.Error.WriteLine(lastError)
         Else
            Console.WriteLine(expandedVariable)
         End If

      End Sub

   End Class

End Namespace
```

```
using System;
using System.Runtime.InteropServices;
using System.Text;

namespace InteroperabilityLibrary
{
   internal class NativeMethods
   {
      private NativeMethods() {}

      // Violates rule UseManagedEquivalentsOfWin32Api.
      [DllImport("kernel32.dll", CharSet = CharSet.Auto,
          SetLastError = true)]
      internal static extern int ExpandEnvironmentStrings(
         string lpSrc, StringBuilder lpDst, int nSize);
   }

   public class UseNativeMethod
   {
      string environmentVariable = "%TEMP%";
      StringBuilder expandedVariable;

      public void ViolateRule()
      {
         expandedVariable = new StringBuilder(100);

         if(NativeMethods.ExpandEnvironmentStrings(
            environmentVariable,
            expandedVariable,
            expandedVariable.Capacity) == 0)
         {
            // Violates rule CallGetLastErrorImmediatelyAfterPInvoke.
            Console.Error.WriteLine(Marshal.GetLastWin32Error());
         }
         else
         {
            Console.WriteLine(expandedVariable);
         }
      }

      public void SatisfyRule()
      {
         expandedVariable = new StringBuilder(100);

         if(NativeMethods.ExpandEnvironmentStrings(
            environmentVariable,
            expandedVariable,
            expandedVariable.Capacity) == 0)
         {
            // Satisfies rule CallGetLastErrorImmediatelyAfterPInvoke.
            int lastError = Marshal.GetLastWin32Error();
            Console.Error.WriteLine(lastError);
         }
         else
         {
            Console.WriteLine(expandedVariable);
         }
      }
   }
}
```

Related rules

CA1060: Move P/Invokes to NativeMethods class

CA1400: P/Invoke entry points should exist

CA1401: P/Invokes should not be visible

CA2101: Specify marshaling for P/Invoke string arguments

CA2205: Use managed equivalents of Win32 API

CA1405: COM visible type base types should be COM visible

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1405
Category	Microsoft.Interoperability
Breaking change	DependsOnFix

Cause

A Component Object Model (COM) visible type derives from a type that is not COM visible.

Rule description

When a COM visible type adds members in a new version, it must abide by strict guidelines to avoid breaking COM clients that bind to the current version. A type that is invisible to COM presumes it does not have to follow these COM versioning rules when it adds new members. However, if a COM visible type derives from the COM invisible type and exposes a class interface of System.Runtime.InteropServices.ClassInterfaceType or ClassInterfaceType (the default), all public members of the base type (unless they are specifically marked as COM invisible, which would be redundant) are exposed to COM. If the base type adds new members in a subsequent version, any COM clients that bind to the class interface of the derived type might break. COM visible types should derive only from COM visible types to reduce the chance of breaking COM clients.

How to fix violations

To fix a violation of this rule, make the base types COM visible or the derived type COM invisible.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a type that violates the rule.

```
Imports System
Imports System.Runtime.InteropServices

<Assembly: ComVisibleAttribute(False)>
Namespace InteroperabilityLibrary

   <ComVisibleAttribute(False)> _
   Public Class BaseClass

      Sub SomeSub(valueOne As Integer)
      End Sub

   End Class

   ' This class violates the rule.
   <ComVisibleAttribute(True)> _
   Public Class DerivedClass
      Inherits BaseClass

      Sub AnotherSub(valueOne As Integer, valueTwo As Integer)
      End Sub

   End Class

End Namespace
```

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(false)]
namespace InteroperabilityLibrary
{
   [ComVisible(false)]
   public class BaseClass
   {
      public void SomeMethod(int valueOne) {}
   }

   // This class violates the rule.
   [ComVisible(true)]
   public class DerivedClass : BaseClass
   {
      public void AnotherMethod(int valueOne, int valueTwo) {}
   }
}
```

See also

- System.Runtime.InteropServices.ClassInterfaceAttribute
- Interoperating with Unmanaged Code

CA1406: Avoid Int64 arguments for Visual Basic 6 clients

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1406
Category	Microsoft.Interoperability
Breaking change	Breaking

Cause

A type that is specifically marked as visible to Component Object Model (COM) declares a member that takes a System.Int64 argument.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Visual Basic 6 COM clients cannot access 64-bit integers.

By default, the following are visible to COM: assemblies, public types, public instance members in public types, and all members of public value types. However, to reduce false positives, this rule requires the COM visibility of the type to be explicitly stated; the containing assembly must be marked with the System.Runtime.InteropServices.ComVisibleAttribute set to `false` and the type must be marked with the ComVisibleAttribute set to `true`.

How to fix violations

To fix a violation of this rule for a parameter whose value can always be expressed as a 32-bit integral, change the parameter type to System.Int32. If the value of the parameter might be larger than can be expressed as a 32-bit integral, change the parameter type to System.Decimal. Note that both System.Single and System.Double lose precision at the upper ranges of the Int64 data type. If the member is not meant to be visible to COM, mark it with the ComVisibleAttribute set to `false`.

When to suppress warnings

It is safe to suppress a warning from this rule if it is certain that Visual Basic 6 COM clients will not access the type.

Example

The following example shows a type that violates the rule.

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(false)]
namespace InteroperabilityLibrary
{
   [ComVisible(true)]
   public class SomeClass
   {
      public void LongArgument(long argument) {}
   }
}
```

```
Imports System
Imports System.Runtime.InteropServices

<Assembly: ComVisibleAttribute(False)>
Namespace InteroperabilityLibrary

   <ComVisibleAttribute(True)> _
   Public Class SomeClass

      Public Sub LongArgument(argument As Long)
      End Sub

   End Class

End Namespace
```

Related rules

CA1413: Avoid non-public fields in COM visible value types

CA1407: Avoid static members in COM visible types

CA1017: Mark assemblies with ComVisibleAttribute

See also

- Interoperating with Unmanaged Code
- Long Data Type

CA1407: Avoid static members in COM visible types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1407
Category	Microsoft.Interoperability
Breaking change	Non-breaking

Cause

A type that is specifically marked as visible to Component Object Model (COM) contains a `public``static` method.

Rule description

COM does not support `static` methods.

This rule ignores property and event accessors, operator overloading methods, or methods that are marked by using either the System.Runtime.InteropServices.ComRegisterFunctionAttribute attribute or the System.Runtime.InteropServices.ComUnregisterFunctionAttribute attribute.

By default, the following are visible to COM: assemblies, public types, public instance members in public types, and all members of public value types.

For this rule to occur, an assembly-level ComVisibleAttribute must be set to `false` and the class-ComVisibleAttribute must be set to `true`, as the following code shows.

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(false)]
namespace Samples
{
    [ComVisible(true)]
    public class MyClass
    {
        public static void DoSomething()
        {
        }
    }
}
```

How to fix violations

To fix a violation of this rule, change the design to use an instance method that provides the same functionality as the `static` method.

When to suppress warnings

It is safe to suppress a warning from this rule if a COM client does not require access to the functionality that is provided by the `static` method.

Example Violation

Description

The following example shows a `static` method that violates this rule.

Code

```
using System;
using System.Runtime.InteropServices;
using System.Collections.ObjectModel;

[assembly: ComVisible(false)]

namespace Samples
{
    [ComVisible(true)]
    public class Book
    {
        private Collection<string> _Pages = new Collection<string>();

        public Book()
        {
        }

        public Collection<string> Pages
        {
            get { return _Pages; }
        }

        // Violates this rule
        public static Book FromPages(string[] pages)
        {
            if (pages == null)
                throw new ArgumentNullException("pages");

            Book book = new Book();

            foreach (string page in pages)
            {
                book.Pages.Add(page);
            }             return book;
        }
    }
}
```

Comments

In this example, the **Book.FromPages** method cannot be called from COM.

Example Fix

Description

To fix the violation in the previous example, you could change the method to an instance method, but that does not make sense in this instance. A better solution is to explicitly apply `ComVisible(false)` to the method to make it clear to other developers that the method cannot be seen from COM.

The following example applies ComRegisterFunctionAttribute to the method.

Code

```
using System;
using System.Runtime.InteropServices;
using System.Collections.ObjectModel;

[assembly: ComVisible(false)]

namespace Samples
{
    [ComVisible(true)]
    public class Book
    {
        private Collection<string> _Pages = new Collection<string>();

        public Book()
        {
        }

        public Collection<string> Pages
        {
            get { return _Pages; }
        }

        [ComVisible(false)]
        public static Book FromPages(string[] pages)
        {
            if (pages == null)
                throw new ArgumentNullException("pages");

            Book book = new Book();

            foreach (string page in pages)
            {
                book.Pages.Add(page);
            }

            return book;
        }
    }
}
```

Related rules

CA1017: Mark assemblies with ComVisibleAttribute

CA1406: Avoid Int64 arguments for Visual Basic 6 clients

CA1413: Avoid non-public fields in COM visible value types

See also

Interoperating with Unmanaged Code

CA1408: Do not use AutoDual ClassInterfaceType

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1408
Category	Microsoft.Interoperability
Breaking change	Breaking

Cause

A Component Object Model (COM) visible type is marked with the ClassInterfaceAttribute attribute set to the `AutoDual` value of ClassInterfaceType.

Rule description

Types that use a dual interface enable clients to bind to a specific interface layout. Any changes in a future version to the layout of the type or any base types will break COM clients that bind to the interface. By default, if the ClassInterfaceAttribute attribute is not specified, a dispatch-only interface is used.

Unless marked otherwise, all public nongeneric types are visible to COM; all nonpublic and generic types are invisible to COM.

How to fix violations

To fix a violation of this rule, change the value of the ClassInterfaceAttribute attribute to the `None` value of ClassInterfaceType and explicitly define the interface.

When to suppress warnings

Do not suppress a warning from this rule unless it is certain that the layout of the type and its base types will not change in a future version.

Example

The following example shows a class that violates the rule and a re-declaration of the class to use an explicit interface.

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(true)]
namespace InteroperabilityLibrary
{
   // This violates the rule.
   [ClassInterface(ClassInterfaceType.AutoDual)]
   public class DualInterface
   {
      public void SomeMethod() {}
   }

   public interface IExplicitInterface
   {
      void SomeMethod();
   }

   [ClassInterface(ClassInterfaceType.None)]
   public class ExplicitInterface : IExplicitInterface
   {
      public void SomeMethod() {}
   }
}
```

```
Imports System
Imports System.Runtime.InteropServices

<Assembly: ComVisibleAttribute(True)>
Namespace InteroperabilityLibrary

   ' This violates the rule.
   <ClassInterfaceAttribute(ClassInterfaceType.AutoDual)> _
   Public Class DualInterface
      Public Sub SomeSub
      End Sub
   End Class

   Public Interface IExplicitInterface
      Sub SomeSub
   End Interface

   <ClassInterfaceAttribute(ClassInterfaceType.None)> _
   Public Class ExplicitInterface
      Implements IExplicitInterface

      Public Sub SomeSub Implements IExplicitInterface.SomeSub
      End Sub

   End Class

End Namespace
```

Related rules

CA1403: Auto layout types should not be COM visible

CA1412: Mark ComSource Interfaces as IDispatch

See also

- Qualifying .NET Types for Interoperation

- Interoperating with Unmanaged Code

CA1409: Com visible types should be creatable

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1409
Category	Microsoft.Interoperability
Breaking change	Non-breaking

Cause

A reference type that is specifically marked as visible to Component Object Model (COM) contains a public parameterized constructor but does not contain a public default (parameterless) constructor.

Rule description

A type without a public default constructor cannot be created by COM clients. However, the type can still be accessed by COM clients if another means is available to create the type and pass it to the client (for example, through the return value of a method call).

The rule ignores types that are derived from System.Delegate.

By default, the following are visible to COM: assemblies, public types, public instance members in public types, and all members of public value types.

How to fix violations

To fix a violation of this rule, add a public default constructor or remove the System.Runtime.InteropServices.ComVisibleAttribute from the type.

When to suppress warnings

It is safe to suppress a warning from this rule if other ways are provided to create and pass the object to the COM client.

Related rules

CA1017: Mark assemblies with ComVisibleAttribute

See also

- Qualifying .NET Types for Interoperation
- Interoperating with Unmanaged Code

CA1410: COM registration methods should be matched

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1410
Category	Microsoft.Interoperability
Breaking change	Non-breaking

Cause

A type declares a method that is marked with the System.Runtime.InteropServices.ComRegisterFunctionAttribute attribute but does not declare a method that is marked with the System.Runtime.InteropServices.ComUnregisterFunctionAttribute attribute, or vice versa.

Rule description

For Component Object Model (COM) clients to create a .NET type, the type must first be registered. If it is available, a method that is marked with the ComRegisterFunctionAttribute attribute is called during the registration process to run user-specified code. A corresponding method that is marked with the ComUnregisterFunctionAttribute attribute is called during the unregistration process to reverse the operations of the registration method.

How to fix violations

To fix a violation of this rule, add the corresponding registration or unregistration method.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a type that violates the rule. The commented code shows the fix for the violation.

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(true)]
namespace InteroperabilityLibrary
{
   public class ClassToRegister
   {
   }

   public class ComRegistration
   {
      [ComRegisterFunction]
      internal static void RegisterFunction(Type typeToRegister) {}

//      [ComUnregisterFunction]
//      internal static void UnregisterFunction(Type typeToRegister) {}
   }
}
```

```
Imports System
Imports System.Runtime.InteropServices

<Assembly: ComVisibleAttribute(True)>
Namespace InteroperabilityLibrary

   Public Class ClassToRegister
   End Class

   Public Class ComRegistration

      <ComRegisterFunctionAttribute> _
      Friend Shared Sub RegisterFunction(typeToRegister As Type)
      End Sub

'      <ComUnregisterFunctionAttribute> _
'      Friend Shared Sub UnregisterFunction(typeToRegister As Type)
'      End Sub

   End Class

End Namespace
```

Related rules

CA1411: COM registration methods should not be visible

See also

- System.Runtime.InteropServices.RegistrationServices
- Register Assemblies with COM
- Regasm.exe (Assembly Registration Tool)

CA1411: COM registration methods should not be visible

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1411
Category	Microsoft.Interoperability
Breaking change	Breaking

Cause

A method that is marked with the System.Runtime.InteropServices.ComRegisterFunctionAttribute or the System.Runtime.InteropServices.ComUnregisterFunctionAttribute attribute is externally visible.

Rule description

When an assembly is registered with Component Object Model (COM), entries are added to the registry for each COM-visible type in the assembly. Methods that are marked with the ComRegisterFunctionAttribute and ComUnregisterFunctionAttribute attributes are called during the registration and unregistration processes, respectively, to run user code that is specific to the registration/unregistration of these types. This code should not be called outside these processes.

How to fix violations

To fix a violation of this rule, change the accessibility of the method to `private` or `internal` (`Friend` in Visual Basic).

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows two methods that violate the rule.

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(true)]
namespace InteroperabilityLibrary
{
   public class ClassToRegister
   {
   }

   public class ComRegistration
   {
      [ComRegisterFunction]
      public static void RegisterFunction(Type typeToRegister) {}

      [ComUnregisterFunction]
      public static void UnregisterFunction(Type typeToRegister) {}
   }
}
```

```
Imports System
Imports System.Runtime.InteropServices

<Assembly: ComVisibleAttribute(True)>
Namespace InteroperabilityLibrary

   Public Class ClassToRegister
   End Class

   Public Class ComRegistration

      <ComRegisterFunctionAttribute> _
      Public Shared Sub RegisterFunction(typeToRegister As Type)
      End Sub

      <ComUnregisterFunctionAttribute> _
      Public Shared Sub UnregisterFunction(typeToRegister As Type)
      End Sub

   End Class

End Namespace
```

Related rules

CA1410: COM registration methods should be matched

See also

- System.Runtime.InteropServices.RegistrationServices
- Registering Assemblies with COM
- Regasm.exe (Assembly Registration Tool)

CA1412: Mark ComSource Interfaces as IDispatch

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1412
Category	Microsoft.Interoperability
Breaking change	Breaking

Cause

A type is marked with the ComSourceInterfacesAttribute attribute and at least one specified interface is not marked with the InterfaceTypeAttribute attribute set to the `InterfaceIsDispatch` value.

Rule description

ComSourceInterfacesAttribute is used to identify the event interfaces that a class exposes to Component Object Model (COM) clients. These interfaces must be exposed as `InterfaceIsIDispatch` to enable Visual Basic 6 COM clients to receive event notifications. By default, if an interface is not marked with the InterfaceTypeAttribute attribute, it is exposed as a dual interface.

How to fix violations

To fix a violation of this rule, add or modify the InterfaceTypeAttribute attribute so that its value is set to InterfaceIsIDispatch for all interfaces that are specified with the ComSourceInterfacesAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a class where one of the interfaces violates the rule.

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(true)]
namespace InteroperabilityLibrary
{
   // This violates the rule for type EventSource.
   [InterfaceType(ComInterfaceType.InterfaceIsDual)]
   public interface IEventsInterface
   {
      void EventOne();
      void EventTwo();
   }

   // This satisfies the rule.
   [InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
   public interface IMoreEventsInterface
   {
      void EventThree();
      void EventFour();
   }

   [ComSourceInterfaces(
      "InteroperabilityLibrary.IEventsInterface\0" +
      "InteroperabilityLibrary.IMoreEventsInterface")]
   public class EventSource
   {
      // Event and method declarations.
   }
}
```

```
Imports Microsoft.VisualBasic
Imports System
Imports System.Runtime.InteropServices

<Assembly: ComVisibleAttribute(True)>
Namespace InteroperabilityLibrary

   ' This violates the rule for type EventSource.
   <InterfaceType(ComInterfaceType.InterfaceIsDual)> _
   Public Interface IEventsInterface
      Sub EventOne
      Sub EventTwo
   End Interface

   ' This satisfies the rule.
   <InterfaceType(ComInterfaceType.InterfaceIsIDispatch)> _
   Public Interface IMoreEventsInterface
      Sub EventThree
      Sub EventFour
   End Interface

   <ComSourceInterfaces( _
      "InteroperabilityLibrary.IEventsInterface" & _
      ControlChars.NullChar & _
      "InteroperabilityLibrary.IMoreEventsInterface")> _
   Public Class EventSource
      ' Event and method declarations.
   End Class

End Namespace
```

Related rules

See also

- Interoperating with Unmanaged Code

CA1413: Avoid non-public fields in COM visible value types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1413
Category	Microsoft.Interoperability
Breaking change	Breaking

Cause

A value type that is specifically marked as visible to Component Object Model (COM) declares a nonpublic instance field.

Rule description

Nonpublic instance fields of COM-visible value types are visible to COM clients. Review the content of the field for information that should not be exposed, or that will have an unintended design or security effect.

By default, all public value types are visible to COM. However, to reduce false positives, this rule requires the COM visibility of the type to be explicitly stated. The containing assembly must be marked with the System.Runtime.InteropServices.ComVisibleAttribute set to `false` and the type must be marked with the ComVisibleAttribute set to `true`.

How to fix violations

To fix a violation of this rule and keep the field hidden, change the value type to a reference type or remove the ComVisibleAttribute attribute from the type.

When to suppress warnings

It is safe to suppress a warning from this rule if public exposure of the field is acceptable.

Example

The following example shows a type that violates the rule.

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(false)]
namespace InteroperabilityLibrary
{
   [ComVisible(true)]
   public struct SomeStruct
   {
      internal int SomeValue;
   }
}
```

```
Imports System
Imports System.Runtime.InteropServices

<Assembly: ComVisibleAttribute(False)>
Namespace InteroperabilityLibrary

   <ComVisibleAttribute(True)> _
   Public Structure SomeStructure

      Friend SomeInteger As Integer

   End Structure

End Namespace
```

Related rules

CA1407: Avoid static members in COM visible types

CA1017: Mark assemblies with ComVisibleAttribute

See also

- Interoperating with Unmanaged Code
- Qualifying .NET Types for Interoperation

CA1414: Mark boolean P/Invoke arguments with MarshalAs

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1414
Category	Microsoft.Interoperability
Breaking change	Breaking

Cause

A platform invoke method declaration includes a System.Boolean parameter or return value but the System.Runtime.InteropServices.MarshalAsAttribute attribute is not applied to the parameter or return value.

Rule description

A platform invoke method accesses unmanaged code and is defined by using the `Declare` keyword in Visual Basic or the System.Runtime.InteropServices.DllImportAttribute. MarshalAsAttribute specifies the marshaling behavior that is used to convert data types between managed and unmanaged code. Many simple data types, such as System.Byte and System.Int32, have a single representation in unmanaged code and do not require specification of their marshaling behavior; the common language runtime automatically supplies the correct behavior.

The Boolean data type has multiple representations in unmanaged code. When the MarshalAsAttribute is not specified, the default marshaling behavior for the Boolean data type is System.Runtime.InteropServices.UnmanagedType. This is a 32-bit integer, which is not appropriate in all circumstances. The Boolean representation that is required by the unmanaged method should be determined and matched to the appropriate System.Runtime.InteropServices.UnmanagedType. UnmanagedType.Bool is the Win32 BOOL type, which is always 4 bytes. UnmanagedType.U1 should be used for C++ `bool` or other 1-byte types.

How to fix violations

To fix a violation of this rule, apply MarshalAsAttribute to the Boolean parameter or return value. Set the value of the attribute to the appropriate UnmanagedType.

When to suppress warnings

Do not suppress a warning from this rule. Even if the default marshaling behavior is appropriate, the code is more easily maintained when the behavior is explicitly specified.

Example

The following example shows platform invoke methods that are marked with the appropriate MarshalAsAttribute attributes.

```
using System;
using System.Runtime.InteropServices;

[assembly: ComVisible(false)]
namespace InteroperabilityLibrary
{
   [ComVisible(true)]
   internal class NativeMethods
   {
      private NativeMethods() {}

      [DllImport("user32.dll", SetLastError = true)]
      [return: MarshalAs(UnmanagedType.Bool)]
      internal static extern Boolean MessageBeep(UInt32 uType);

      [DllImport("mscoree.dll",
                 CharSet = CharSet.Unicode,
                 SetLastError = true)]
      [return: MarshalAs(UnmanagedType.U1)]
      internal static extern bool StrongNameSignatureVerificationEx(
         [MarshalAs(UnmanagedType.LPWStr)] string wszFilePath,
         [MarshalAs(UnmanagedType.U1)] bool fForceVerification,
         [MarshalAs(UnmanagedType.U1)] out bool pfWasVerified);
   }
}
```

```
Imports System
Imports System.Runtime.InteropServices

<assembly: ComVisible(False)>
Namespace UsageLibrary

   <ComVisible(True)> _
   Class NativeMethods

      Private Sub New()
      End Sub

      <DllImport("user32.dll", SetLastError := True)> _
      Friend Shared Function MessageBeep(uType As UInt32) _
         As <MarshalAs(UnmanagedType.Bool)> Boolean
      End Function

      <DllImport("mscoree.dll", SetLastError := True)> _
      Friend Shared Function StrongNameSignatureVerificationEx( _
         <MarshalAs(UnmanagedType.LPWStr)> wszFilePath As String, _
         <MarshalAs(UnmanagedType.U1)> fForceVerification As Boolean, _
         <MarshalAs(UnmanagedType.U1)> ByRef pfWasVerified As Boolean) _
         As <MarshalAs(UnmanagedType.U1)> Boolean
      End Function

   End Class

End Namespace
```

```
using namespace System;
using namespace System::Runtime::InteropServices;

[assembly: ComVisible(false)];
namespace InteroperabilityLibrary
{
   [ComVisible(true)]
   ref class NativeMethods
   {
   private:
      NativeMethods() {}

   internal:
      [DllImport("user32.dll", SetLastError = true)]
      [returnvalue: MarshalAs(UnmanagedType::Bool)]
      static Boolean MessageBeep(UInt32 uType);

      [DllImport("mscoree.dll",
                 CharSet = CharSet::Unicode,
                 SetLastError = true)]
      [returnvalue: MarshalAs(UnmanagedType::U1)]
      static bool StrongNameSignatureVerificationEx(
         [MarshalAs(UnmanagedType::LPWStr)] String^ wszFilePath,
         [MarshalAs(UnmanagedType::U1)] Boolean fForceVerification,
         [MarshalAs(UnmanagedType::U1)] Boolean^ pfWasVerified);
   };
}
```

Related rules

CA1901: P/Invoke declarations should be portable

CA2101: Specify marshaling for P/Invoke string arguments

See also

- System.Runtime.InteropServices.UnmanagedType
- Interoperating with Unmanaged Code

CA1415: Declare P/Invokes correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1415
Category	Microsoft.Interoperability
Breaking change	Non-breaking - If the P/Invoke that declares the parameter cannot be seen outside the assembly. Breaking - If the P/Invoke that declares the parameter can be seen outside the assembly.

Cause

A platform invoke method is incorrectly declared.

Rule description

A platform invoke method accesses unmanaged code and is defined by using the `Declare` keyword in Visual Basic or the System.Runtime.InteropServices.DllImportAttribute. Currently, this rule looks for platform invoke method declarations that target Win32 functions that have a pointer to an OVERLAPPED structure parameter and the corresponding managed parameter is not a pointer to a System.Threading.NativeOverlapped structure.

How to fix violations

To fix a violation of this rule, correctly declare the platform invoke method.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows platform invoke methods that violate the rule and satisfy the rule.

```
using System;
using System.Runtime.InteropServices;
using System.Threading;

namespace InteroperabilityLibrary
{
    // The platform invoke methods in this class violate the rule.
    [ComVisible(true)]
    internal class NativeMethods
    {
        private NativeMethods() { }

        [DllImport("kernel32.dll", SetLastError = true)]
        internal extern static uint ReadFile(
           IntPtr hFile, IntPtr lpBuffer, int nNumberOfBytesToRead,
           IntPtr lpNumberOfBytesRead, IntPtr overlapped);

        [DllImport("kernel32.dll", SetLastError = true)]
        [return: MarshalAs(UnmanagedType.Bool)]
        internal extern static bool ReadFileEx(
           IntPtr hFile, IntPtr lpBuffer, int nNumberOfBytesToRead,
           NativeOverlapped overlapped, IntPtr lpCompletionRoutine);
    }

    // The platform invoke methods in this class satisfy the rule.
    [ComVisible(true)]
    internal class UnsafeNativeMethods
    {
        private UnsafeNativeMethods() { }

        //To compile this code, uncomment these lines and compile
        //with "/unsafe".
        //[DllImport("kernel32.dll", SetLastError = true)]
        //unsafe internal extern static uint ReadFile(
        //   IntPtr hFile, IntPtr lpBuffer, int nNumberOfBytesToRead,
        //   IntPtr lpNumberOfBytesRead, NativeOverlapped* overlapped);

        //[DllImport("kernel32.dll", SetLastError = true)]
        //[return: MarshalAs(UnmanagedType.Bool)]
        //unsafe internal extern static bool ReadFileEx(
        //   IntPtr hFile, IntPtr lpBuffer, int nNumberOfBytesToRead,
        //   NativeOverlapped* overlapped, IntPtr lpCompletionRoutine);
    }
}
```

See also

Interoperating with Unmanaged Code

CA1417: Do not use `OutAttribute` on string parameters for P/Invokes

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1417
Category	Microsoft.Interoperability
Breaking change	Non-breaking

Cause

A P/Invoke string parameter is passed by value and marked with OutAttribute.

Rule description

The .NET runtime automatically performs string interning. If an interned string marked with OutAttribute is passed by value to a P/Invoke, the runtime can be destabilized.

How to fix violations

If marshaling of modified string data back to the caller is required, pass the string by reference instead. Otherwise, the OutAttribute can be removed without any other changes.

```
 // Violation
[DllImport("MyLibrary")]
private static extern void Foo([Out] string s);

// Fixed: passed by reference
[DllImport("MyLibrary")]
private static extern void Foo(out string s);

// Fixed: marshaling data back to caller is not required
[DllImport("MyLibrary")]
private static extern void Foo(string s);
```

When to suppress warnings

It is not safe to suppress a warning from this rule.

See also

- Interoperability warnings
- Native interoperability best practices

Maintainability warnings

4/30/2020 • 2 minutes to read • Edit Online

Maintainability warnings support library and application maintenance.

In this section

RULE	DESCRIPTION
CA1500: Variable names should not match field names	An instance method declares a parameter or a local variable whose name matches an instance field of the declaring type, which leads to errors.
CA1501: Avoid excessive inheritance	A type is more than four levels deep in its inheritance hierarchy. Deeply nested type hierarchies can be difficult to follow, understand, and maintain.
CA1502: Avoid excessive complexity	This rule measures the number of linearly independent paths through the method, which is determined by the number and complexity of conditional branches.
CA1504: Review misleading field names	The name of an instance field starts with "s_", or the name of a static (Shared in Visual Basic) field starts with "m_".
CA1505: Avoid unmaintainable code	A type or method has a low maintainability index value. A low maintainability index indicates that a type or method is probably difficult to maintain and would be a good candidate for redesign.
CA1506: Avoid excessive class coupling	This rule measures class coupling by counting the number of unique type references that a type or method contains.
CA1507: Use nameof in place of string	A string literal is used as an argument where a `nameof` expression could be used.
CA1508: Avoid dead conditional code	A method has conditional code that always evaluates to `true` or `false` at runtime. This leads to dead code in the `false` branch of the condition.
CA1509: Invalid entry in code metrics configuration file	Code metrics rules, such as CA1501, CA1502, CA1505 and CA1506, supplied a configuration file named `CodeMetricsConfig.txt` that has an invalid entry.

See also

- Measure Complexity and Maintainability of Managed Code

CA1500: Variable names should not match field names

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1500
Category	Microsoft.Maintainability
Breaking change	When fired on a parameter that has the same name as a field: - Non-breaking - If both the field and method that declares the parameter cannot be seen outside the assembly, regardless of the change you make. - Breaking - If you change the name of the field and can be seen outside the assembly. - Breaking - If you change the name of the parameter and the method that declares it can be seen outside the assembly. When fired on a local variable that has the same name as a field: - Non-breaking - If the field cannot be seen outside the assembly, regardless of the change you make. - Non-breaking - If you change the name of the local variable and do not change the name of the field. - Breaking - If you change the name of the field and it can be seen outside the assembly.

Cause

An instance method declares a parameter or a local variable whose name matches an instance field of the declaring type. To catch local variables that violate the rule, the tested assembly must be built by using debugging information and the associated program database (.pdb) file must be available.

Rule description

When the name of an instance field matches a parameter or a local variable name, the instance field is accessed by using the `this` (`Me` in Visual Basic) keyword when inside the method body. When maintaining code, it is easy to forget this difference and assume that the parameter/local variable refers to the instance field, which leads to errors. This is true especially for lengthy method bodies.

How to fix violations

To fix a violation of this rule, rename either the parameter/variable or the field.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows two violations of the rule.

```
Imports System

Namespace MaintainabilityLibrary

   Class MatchingNames

      Dim someField As Integer

      Sub SomeMethodOne(someField As Integer)
      End Sub

      Sub SomeMethodTwo()
         Dim someField As Integer
      End Sub

   End Class

End Namespace
```

```
using System;

namespace MaintainabilityLibrary
{
   class MatchingNames
   {
      int someField;

      void SomeMethodOne(int someField) {}

      void SomeMethodTwo()
      {
         int someField;
      }
   }
}
```

CA1501: Avoid excessive inheritance

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1501
Category	Microsoft.Maintainability
Breaking change	Breaking

Cause

A type is more than four levels deep in its inheritance hierarchy.

By default, the rule only excludes all types from the `System` namespace, but this is configurable.

Rule description

Deeply nested type hierarchies can be difficult to follow, understand, and maintain. This rule limits analysis to hierarchies in the same module.

How to fix violations

To fix a violation of this rule, derive the type from a base type that is less deep in the inheritance hierarchy or eliminate some of the intermediate base types.

When to suppress warnings

It is safe to suppress a warning from this rule. However, the code might be more difficult to maintain. Note that, depending on the visibility of base types, resolving violations of this rule might create breaking changes. For example, removing public base types is a breaking change.

Configurability

Inheritance excluded type or namespace names

You can configure names of types or namespaces (separated by '|'), such that the type or type's namespace doesn't count in the inheritance hierarchy tree. By default, all types from the `System.*` namespace are excluded. Note that no matter which value you are setting, this default value will be added.

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = MyType`	Matches all types named 'MyType' or whose containing namespace contains 'MyType' and all types from the 'System' namespace
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = MyType1|MyType2`	Matches all types named either 'MyType1' or 'MyType2' or whose containing namespace contains either 'MyType1' or 'MyType2' and all types from the 'System' namespace

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA1501.additional_inheritance_excluded_types_specific = T:NS.MyType`	Matches specific type 'MyType' in the namespace 'NS' and all types from the 'System' namespace
`dotnet_code_quality.CA1501.additional_inheritance_excluded_types_specific = T:NS1.MyType1|T:NS2.MyType2`	Matches specific types 'MyType1' and 'MyType2' with respective fully qualified names and all types from the 'System' namespace
`dotnet_code_quality.CA1501.additional_inheritance_excluded_types_all_types = N:NS`	Matches all types from the 'NS' namespace and all types from the 'System' namespace
`dotnet_code_quality.CA1501.additional_inheritance_excluded_types_all_types = My*`	Matches all types whose name starts with 'My' or whose containing namespace parts starts with 'My' and all types from the 'System' namespace
`dotnet_code_quality.CA1501.additional_inheritance_excluded_types_all_types = T:NS.My*`	Matches all types whose name starts with 'My' in the namespace 'NS' and all types from the 'System' namespace
`dotnet_code_quality.CA1501.additional_inheritance_excluded_types_all_types = N:My*`	Matches all types whose containing namespace starts with 'My' and all types from the 'System' namespace

Example

The following example shows a type that violates the rule:

```
using System;

namespace MaintainabilityLibrary
{
   class BaseClass {}
   class FirstDerivedClass : BaseClass {}
   class SecondDerivedClass : FirstDerivedClass {}
   class ThirdDerivedClass : SecondDerivedClass {}
   class FourthDerivedClass : ThirdDerivedClass {}

   // This class violates the rule.
   class FifthDerivedClass : FourthDerivedClass {}
}
```

```
Imports System

Namespace MaintainabilityLibrary

   Class BaseClass
   End Class

   Class FirstDerivedClass
      Inherits BaseClass
   End Class

   Class SecondDerivedClass
      Inherits FirstDerivedClass
   End Class

   Class ThirdDerivedClass
      Inherits SecondDerivedClass
   End Class

   Class FourthDerivedClass
      Inherits ThirdDerivedClass
   End Class

   ' This class violates the rule.
   Class FifthDerivedClass
      Inherits FourthDerivedClass
   End Class

End Namespace
```

CA1502: Avoid excessive complexity

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1502
Category	Microsoft.Maintainability
Breaking change	Non-breaking

Cause

A method has an excessive cyclomatic complexity.

Rule description

Cyclomatic complexity measures the number of linearly independent paths through the method, which is determined by the number and complexity of conditional branches. A low cyclomatic complexity generally indicates a method that is easy to understand, test, and maintain. The cyclomatic complexity is calculated from a control flow graph of the method and is given as follows:

cyclomatic complexity = the number of edges - the number of nodes + 1

A *node* represents a logic branch point and an *edge* represents a line between nodes.

The rule reports a violation when the cyclomatic complexity is more than 25.

You can learn more about code metrics at Measure complexity of managed code.

How to fix violations

To fix a violation of this rule, refactor the method to reduce its cyclomatic complexity.

When to suppress warnings

It is safe to suppress a warning from this rule if the complexity cannot easily be reduced and the method is easy to understand, test, and maintain. In particular, a method that contains a large `switch` (`Select` in Visual Basic) statement is a candidate for exclusion. The risk of destabilizing the code base late in the development cycle or introducing an unexpected change in run-time behavior in previously shipped code might outweigh the maintainability benefits of refactoring the code.

How Cyclomatic complexity is calculated

The cyclomatic complexity is calculated by adding 1 to the following:

- Number of branches (such as `if`, `while`, and `do`)
- Number of `case` statements in a `switch`

Example

The following examples show methods that have varying cyclomatic complexities.

Cyclomatic complexity of 1

```
void Method()
{
    Console::WriteLine("Hello World!");
}
```

```
Public Sub Method()
    Console.WriteLine("Hello World!")
End Sub
```

```
public void Method()
{
    Console.WriteLine("Hello World!");
}
```

Example

Cyclomatic complexity of 2

```
void Method(bool condition)
{
  if (condition)
    {
        Console::WriteLine("Hello World!");
    }
}
```

```
Public Sub Method(ByVal condition As Boolean)
    If (condition) Then
        Console.WriteLine("Hello World!")
    End If
End Sub
```

```
void Method(bool condition)
{
    if (condition)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Example

Cyclomatic complexity of 3

```
void Method(bool condition1, bool condition2)
{
    if (condition1 || condition2)
    {
        Console::WriteLine("Hello World!");
    }
}
```

```
Public Sub Method(ByVal condition1 As Boolean, ByVal condition2 As Boolean)
    If (condition1 OrElse condition2) Then
        Console.WriteLine("Hello World!")
    End If
End Sub
```

```
public void Method(bool condition1, bool condition2)
{
    if (condition1 || condition2)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Example

Cyclomatic complexity of 8

```
void Method(DayOfWeek day)
{
    switch (day)
    {
        case DayOfWeek::Monday:
            Console::WriteLine("Today is Monday!");
            break;
        case DayOfWeek::Tuesday:
            Console::WriteLine("Today is Tuesday!");
        break;
        case DayOfWeek::Wednesday:
            Console::WriteLine("Today is Wednesday!");
        break;
        case DayOfWeek::Thursday:
            Console::WriteLine("Today is Thursday!");
        break;
        case DayOfWeek::Friday:
            Console::WriteLine("Today is Friday!");
        break;
        case DayOfWeek::Saturday:
            Console::WriteLine("Today is Saturday!");
        break;
        case DayOfWeek::Sunday:
            Console::WriteLine("Today is Sunday!");
        break;
    }
}
```

```
Public Sub Method(ByVal day As DayOfWeek)
    Select Case day
        Case DayOfWeek.Monday
            Console.WriteLine("Today is Monday!")
        Case DayOfWeek.Tuesday
            Console.WriteLine("Today is Tuesday!")
        Case DayOfWeek.Wednesday
            Console.WriteLine("Today is Wednesday!")
        Case DayOfWeek.Thursday
            Console.WriteLine("Today is Thursday!")
        Case DayOfWeek.Friday
            Console.WriteLine("Today is Friday!")
        Case DayOfWeek.Saturday
            Console.WriteLine("Today is Saturday!")
        Case DayOfWeek.Sunday
            Console.WriteLine("Today is Sunday!")
    End Select
End Sub
```

```
    public void Method(DayOfWeek day)
    {
        switch (day)
        {
            case DayOfWeek.Monday:
                Console.WriteLine("Today is Monday!");
                break;
            case DayOfWeek.Tuesday:
                Console.WriteLine("Today is Tuesday!");
                break;
            case DayOfWeek.Wednesday:
                Console.WriteLine("Today is Wednesday!");
                break;
            case DayOfWeek.Thursday:
                Console.WriteLine("Today is Thursday!");
                break;
            case DayOfWeek.Friday:
                Console.WriteLine("Today is Friday!");
                break;
            case DayOfWeek.Saturday:
                Console.WriteLine("Today is Saturday!");
                break;
            case DayOfWeek.Sunday:
                Console.WriteLine("Today is Sunday!");
                break;
        }
    }
}
```

Related rules

CA1501: Avoid excessive inheritance

See also

- Measuring Complexity and Maintainability of Managed Code

CA1504: Review misleading field names

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1504
Category	Microsoft.Maintainability
Breaking change	Non-breaking

Cause

The name of an instance field starts with "s_" or the name of a `static` (`Shared` in Visual Basic) field starts with "m_".

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Field names that start with "s_" are associated with static data by many users. Similarly, field names that start with "m_" are associated with instance (member) data. For more easily maintained code, names should follow generally used conventions.

How to fix violations

To fix a violation of this rule, rename the field by using the appropriate prefix. Alternatively, make the field agree with the current suffix by adding or removing the `static` modifier.

When to suppress warnings

Do not suppress a warning from this rule.

CA1505: Avoid unmaintainable code

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1505
Category	Microsoft.Maintainability
Breaking change	Non-breaking

Cause

A type or method has a low maintainability index value.

Rule description

The maintainability index is calculated by using the following metrics: lines of code, program volume, and cyclomatic complexity. Program volume is a measure of the difficulty of understanding of a type or method that's based on the number of operators and operands in the code. Cyclomatic complexity is a measure of the structural complexity of the type or method. You can learn more about code metrics at Measure complexity and maintainability of managed code.

A low maintainability index indicates that a type or method is probably difficult to maintain and would be a good candidate to redesign.

How to fix violations

To fix this violation, redesign the type or method and try to split it into smaller and more focused types or methods.

When to suppress warnings

You can suppress this warning when the type or method cannot be split or is considered maintainable despite its large size.

See also

- Maintainability warnings
- Measure complexity and maintainability of managed code

CA1506: Avoid excessive class coupling

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1506
Category	Microsoft.Maintainability
Breaking change	Breaking

Cause

A type or method is coupled with many other types. Compiler generated types are excluded from this metric.

Rule description

This rule measures class coupling by counting the number of unique type references that a type or method contains.

Types and methods that have a high degree of class coupling can be difficult to maintain. It's a good practice to have types and methods that exhibit low coupling and high cohesion.

How to fix violations

To fix this violation, try to redesign the type or method to reduce the number of types to which it's coupled.

When to suppress warnings

Exclude this warning when the type or method is considered maintainable despite its large number of dependencies on other types.

See also

- Maintainability Warnings
- Measuring Complexity and Maintainability of Managed Code

CA1507: Use `nameof` in place of string

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1507
Category	Microsoft.Maintainability
Breaking change	Non-breaking

Cause

A `string` literal or constant that matches the name of a parameter of the containing method or the name of a property of the containing type is used as an argument to a method.

Rule description

Rule CA1507 flags the use of a `string` literal as an argument to a method or constructor where a nameof (`NameOf` in Visual Basic) expression would add maintainability. The rule fires if all of the following conditions are met:

- The argument is a `string` literal or constant.
- The argument corresponds to a `string`-typed parameter of the method or the constructor that's being invoked (that is, there is no conversion involved at the call site).
- Either:
  - The declared name of the parameter is `paramName` and the constant value of the `string` literal matches the name of a parameter of the method, lambda, or local function within which the method or constructor is being invoked.
  - The declared name of the parameter is `propertyName` and the constant value of the `string` literal matches the name of a property of the type within which the method or constructor is being invoked.

Rule CA1507 improves code maintainability in cases where the parameter may be renamed in the future, but the `string` literal is mistakenly not renamed. By using `nameof`, the symbol will be renamed when the parameter is renamed through a refactoring operation. In addition, any spelling mistakes in the name of the parameter are caught by the compiler.

> **NOTE**
>
> Rule CA1507 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.6.0.

How to fix violations

To fix a violation, replace the `string` literal with a nameof (`NameOf` in Visual Basic) expression. For example, the following two code snippets show a violation of the rule and how to fix it:

```
public Book(string title)
{
    // Violates rule CA1507
    Title = title ?? throw new ArgumentNullException("title", "All books must have a title.");
}
```

```
public Book(string title)
{
    // Resolves rule CA1507 violation
    Title = title ?? throw new ArgumentNullException(nameof(title), "All books must have a title.");
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the `string` literal and press **Ctrl+.** (period). Choose **Use nameof to express symbol names** from the list of options that's presented.

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about the maintainability of your code.

Related rules

- CA2208: Instantiate argument exceptions correctly

See also

- Maintainability warnings

CA1508: Avoid dead conditional code

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1508
Category	Microsoft.Maintainability
Breaking change	Non-Breaking

Cause

A method has conditional code that always evaluates to `true` or `false` at runtime. This leads to dead code in the `false` branch of the condition.

Rule description

Methods can have conditional code, such as if statements, binary expressions (`==`, `!=`, `<`, `>`), null checks, etc. For example, consider the following code:

```
public void M(int i, int j)
{
    if (i != 0)
    {
        return;
    }

    if (j != 0)
    {
        return;
    }

    // Below condition will always evaluate to 'false' as 'i' and 'j' are both '0' here.
    if (i != j)
    {
        // Code in this 'if' branch is dead code.
        // It can either be removed or refactored.
        ...
    }
}
```

C# and VB compilers perform analysis of conditional checks involving compile-time *constant values* that always evaluate to `true` or `false`. This analyzer performs data flow analysis of non-constant variables to determine redundant conditional checks involving *non-constant values*. In the preceding code, the analyzer determines that `i` and `j` are both `0` for all code paths that reach `i != j` check. Hence, this check will always evaluate to `false` at runtime. The code inside the if statement is dead code and can be removed or refactored. Similarly, the analyzer tracks nullness of variables and reports redundant null checks.

> **NOTE**
>
> Rule CA1508 is not available in legacy analysis. It was first introduced in FxCop analyzers. Note that this analyzer performs an expensive dataflow analysis of non-constant values. This can increase the overall compile time on certain code bases. Hence, this rule is disabled by default in the analyzer package. Users can enable this rule in editorconfig and perform a one-off analysis and code cleanup. For more information on enabling the rule in editorconfig, see here.

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about the maintainability of your code. It is also fine to suppress violations that are identified to be false positives. These are possible in the presence of concurrent code that can execute from multiple threads.

See also

- Maintainability warnings

CA1509: Invalid entry in code metrics configuration file

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1509
Category	Microsoft.Maintainability
Breaking change	Non-Breaking

Cause

Code metrics rules, such as CA1501, CA1502, CA1505 and CA1506, supplied a configuration file named `CodeMetricsConfig.txt` that has an invalid entry.

Rule description

FxCop analyzers implementation of code metrics analysis rules allow end users to supply an additional file named `CodeMetricsConfig.txt`. This file contains entries to configure code metric thresholds for analysis. Following rules are configurable in this file:

- CA1501: Avoid excessive inheritance
- CA1502: Avoid excessive complexity
- CA1505: Avoid unmaintainable code
- CA1506: Avoid excessive class coupling

This configuration file expects each entry to be in following format:

```
'RuleId'(Optional 'SymbolKind'): 'Threshold'
```

- Valid values for 'RuleId' are `CA1501`, `CA1502`, `CA1505`, and `CA1506`.
- Valid values for optional 'SymbolKind' are `Assembly`, `Namespace`, `Type`, `Method`, `Field`, `Event`, and `Property`.
- Valid values for 'Threshold' are non-negative integers.
- Lines starting with '#' are treated as comment lines

For example, the following is a valid configuration file:

```
# Comment text

CA1501: 1

CA1502(Type): 4
CA1502(Method): 2
```

An invalid entry in this configuration file is flagged with the `CA1509` diagnostic.

> **NOTE**
> Rule CA1509 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.6.

How to fix violations

To fix a violation of this rule, make sure the invalid entry in `CodeMetricsConfig.txt` gets the required format.

When to suppress warnings

Do not suppress violations of this rule.

Related rules

- CA1501: Avoid excessive inheritance
- CA1502: Avoid excessive complexity
- CA1505: Avoid unmaintainable code
- CA1506: Avoid excessive class coupling

See also

- Maintainability warnings
- Measure complexity and maintainability of managed code

Mobility Warnings

2/28/2020 • 2 minutes to read • Edit Online

Mobility warnings support efficient power usage.

In This Section

RULE	DESCRIPTION
CA1600: Do not use idle process priority	Do not set process priority to Idle. Processes that have System.Diagnostics.ProcessPriorityClass.Idle will occupy the CPU when it would otherwise be idle, and will therefore block standby.
CA1601: Do not use timers that prevent power state changes	Higher-frequency periodic activity will keep the CPU busy and interfere with power-saving idle timers that turn off the display and hard disks.

CA1600: Do not use idle process priority

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1600
Category	Microsoft.Mobility
Breaking change	Breaking

Cause

This rule occurs when processes are set to `ProcessPriorityClass.Idle`.

Rule description

Do not set process priority to Idle. Processes that have `System.Diagnostics.ProcessPriorityClass.Idle` will occupy the CPU when it would otherwise be idle, and will therefore block standby.

How to fix violations

Set processes to `ProcessPriorityClass.BelowNormal`.

When to suppress warnings

This rule should be suppressed only when Idle process priority is required and mobility considerations can be ignored safely.

CA1601: Do not use timers that prevent power state changes

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1601
Category	Microsoft.Mobility
Breaking change	Breaking

Cause

A timer has an interval set to occur more than one time per second.

Rule description

Do not poll more often than one time per second or use timers that occur more frequently than one time per second. Higher-frequency periodic activity will keep the CPU busy and interfere with power-saving idle timers that turn off the display and hard disks.

How to fix violations

Set timer intervals to occur less than one time per second.

When to suppress warnings

This rule should be suppressed only if firing the timer more than one time per second is required and mobility considerations can safely be ignored.

Naming Warnings

1/1/2020 • 4 minutes to read • Edit Online

Naming warnings support adherence to the naming conventions of the .NET Design Guidelines.

In This Section

RULE	DESCRIPTION
CA1700: Do not name enum values 'Reserved'	This rule assumes that an enumeration member that has a name that contains "reserved" is not currently used but is a placeholder to be renamed or removed in a future version. Renaming or removing a member is a breaking change.
CA1713: Events should not have before or after prefix	The name of an event starts with "Before" or "After". To name related events that are raised in a specific sequence, use the present or past tense to indicate the relative position in the sequence of actions.
CA1714: Flags enums should have plural names	A public enumeration has the System.FlagsAttribute attribute and its name does not end in "s". Types that are marked with FlagsAttribute have names that are plural because the attribute indicates that more than one value can be specified.
CA1704: Identifiers should be spelled correctly	The name of an externally visible identifier contains one or more words that are not recognized by the Microsoft spelling checker library.
CA1708: Identifiers should differ by more than case	Identifiers for namespaces, types, members, and parameters cannot differ only by case because languages that target the common language runtime are not required to be case-sensitive.
CA1715: Identifiers should have correct prefix	The name of an externally visible interface does not start with a capital "I". The name of a generic type parameter on an externally visible type or method does not start with a capital "T".
CA1720: Identifiers should not contain type names	The name of a parameter in an externally visible member contains a data type name, or the name of an externally visible member contains a language-specific data type name.
CA1722: Identifiers should not have incorrect prefix	By convention, only certain programming elements have names that begin with a specific prefix.
CA1711: Identifiers should not have incorrect suffix	By convention, only the names of types that extend certain base types or that implement certain interfaces, or types that are derived from these types, should end with specific reserved suffixes. Other type names should not use these reserved suffixes.

RULE	DESCRIPTION
CA1717: Only FlagsAttribute enums should have plural names	Naming conventions dictate that a plural name for an enumeration indicates that more than one value of the enumeration can be specified at the same time.
CA1725: Parameter names should match base declaration	Consistent naming of parameters in an override hierarchy increases the usability of the method overrides. A parameter name in a derived method that differs from the name in the base declaration can cause confusion about whether the method is an override of the base method or a new overload of the method.
CA1719: Parameter names should not match member names	A parameter name should communicate the meaning of a parameter, and a member name should communicate the meaning of a member. It would be a rare design where these were the same. Naming a parameter the same as its member name is unintuitive and makes the library difficult to use.
CA1701: Resource string compound words should be cased correctly	Each word in the resource string is split into tokens that are based on the casing. Each contiguous two-token combination is checked by the Microsoft spelling checker library. If recognized, the word produces a violation of the rule.
CA1703: Resource strings should be spelled correctly	A resource string contains one or more words that are not recognized by the Microsoft spelling checker library.
CA1724: Type Names Should Not Match Namespaces	Type names should not match the names of .NET namespaces. Violation of this rule can reduce the usability of the library.
CA1707: Identifiers should not contain underscores	By convention, identifier names do not contain the underscore (_) character. This rule checks namespaces, types, members, and parameters.
CA1721: Property names should not match get methods	The name of a public or protected member starts with "Get" and otherwise matches the name of a public or protected property. "Get" methods and properties should have names that clearly distinguish their function.
CA1716: Identifiers should not match keywords	A namespace name or a type name matches a reserved keyword in a programming language. Identifiers for namespaces and types should not match keywords that are defined by languages that target the common language runtime.
CA1726: Use preferred terms	The name of an externally visible identifier includes a term for which an alternative, preferred term exists. Alternatively, the name includes the term "Flag" or "Flags".
CA1709: Identifiers should be cased correctly	By convention, parameter names use camel casing, and namespace, type, and member names use Pascal casing.
CA1702: Compound words should be cased correctly	The name of an identifier contains multiple words, and at least one of the words appears to be a compound word that is not cased correctly.

RULE	DESCRIPTION
CA1712: Do not prefix enum values with type name	Names of enumeration members are not prefixed with the type name because type information is expected to be provided by development tools.
CA1710: Identifiers should have correct suffix	By convention, the names of types that extend certain base types or that implement certain interfaces, or types derived from these types, have a suffix that is associated with the base type or interface.

CA1700: Do not name enum values 'Reserved'

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1700
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The name of an enumeration member contains the word "reserved".

Rule description

This rule assumes that an enumeration member that has a name that contains "reserved" is not currently used but is a placeholder to be renamed or removed in a future version. Renaming or removing a member is a breaking change. You should not expect users to ignore a member just because its name contains "reserved", nor can you rely on users to read or abide by documentation. Furthermore, because reserved members appear in object browsers and smart integrated development environments, they can cause confusion about which members are actually being used.

Instead of using a reserved member, add a new member to the enumeration in the future version. In most cases the addition of the new member is not a breaking change, as long as the addition does not cause the values of the original members to change.

In a limited number of cases the addition of a member is a breaking change even when the original members retain their original values. Primarily, the new member cannot be returned from existing code paths without breaking callers that use a `switch` (`Select` in Visual Basic) statement on the return value that encompasses the whole member list and that throw an exception in the default case. A secondary concern is that client code might not handle the change in behavior from reflection methods such as System.Enum.IsDefined. Accordingly, if the new member has to be returned from existing methods or a known application incompatibility occurs because of poor reflection usage, the only nonbreaking solution is to:

1. Add a new enumeration that contains the original and new members.
2. Mark the original enumeration with the System.ObsoleteAttribute attribute.

   Follow the same procedure for any externally visible types or members that expose the original enumeration.

How to fix violations

To fix a violation of this rule, remove or rename the member.

When to suppress warnings

It is safe to suppress a warning from this rule for a member that is currently used or for libraries that have previously shipped.

Related rules

CA2217: Do not mark enums with FlagsAttribute

CA1712: Do not prefix enum values with type name

CA1028: Enum storage should be Int32

CA1008: Enums should have zero value

CA1027: Mark enums with FlagsAttribute

CA1701: Resource string compound words should be cased correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1701
Category	Microsoft.Naming
Breaking change	Non-breaking

Cause

A resource string contains a compound word that does not appear to be cased correctly.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Each word in the resource string is split into tokens that are based on the casing. Each contiguous two-token combination is checked by the Microsoft spelling checker library. If recognized, the word produces a violation of the rule. Examples of compound words that cause a violation are "CheckSum" and "MultiPart", which should be cased as "Checksum" and "Multipart", respectively. Due to previous common usage, several exceptions are built into the rule, and several single words are flagged, such as "Toolbar" and "Filename", that should be cased as two distinct words. In this example, "ToolBar" and "FileName" would be flagged.

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the learning curve that is required for new software libraries, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

Change the word so that it is cased correctly.

Change the dictionary language

By default, the English (en) version of the spelling checker is used. If you want to change the language of the spelling checker, you can do so by adding one of the following attributes to your *AssemblyInfo.cs* or *AssemblyInfo.vb* file:

- Use AssemblyCultureAttribute to specify the culture if your resources are in a satellite assembly.
- Use NeutralResourcesLanguageAttribute to specify the *neutral culture* of your assembly if your resources are in the same assembly as your code.

> **IMPORTANT**
>
> If you set the culture to anything other than an English-based culture, this code analysis rule is silently disabled.

When to suppress warnings

It is safe to suppress a warning from this rule if both parts of the compound word are recognized by the spelling dictionary and the intent is to use two words.

You can also add compound words to a custom dictionary for the spelling checker. Words in the custom dictionary do not cause violations. For more information, see How to: Customize the Code Analysis Dictionary.

Related rules

- CA1702: Compound words should be cased correctly
- CA1709: Identifiers should be cased correctly
- CA1708: Identifiers should differ by more than case

See also

- Capitalization Conventions
- Naming Guidelines

CA1702: Compound words should be cased correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1702
Category	Microsoft.Naming
Breaking change	Breaking- when fired on assemblies. Non-breaking - when fired on type parameters.

Cause

The name of an identifier contains multiple words, and at least one of the words appears to be a compound word that is not cased correctly.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

The name of the identifier is split into words that are based on the casing. Each contiguous two-word combination is checked by the Microsoft spelling checker library. If it is recognized, the identifier produces a violation of the rule. Examples of compound words that cause a violation are "CheckSum" and "MultiPart", which should be cased as "Checksum" and "Multipart", respectively. Due to previous common usage, several exceptions are built into the rule, and several single words are flagged, such as "Toolbar" and "Filename", that should be cased as two distinct words (in this case, "ToolBar" and "FileName").

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the learning curve that is required for new software libraries, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

Change the name so that it is cased correctly.

Language

The spell checker currently checks only against English-based culture dictionaries. You can change the culture of your project in the project file, by adding the **CodeAnalysisCulture** element.

For example:

```
<Project ...>
  <PropertyGroup>
    <CodeAnalysisCulture>en-AU</CodeAnalysisCulture>
```

> **IMPORTANT**
>
> If you set the culture to anything other than an English-based culture, this code analysis rule is silently disabled.

When to suppress warnings

It is safe to suppress a warning from this rule if both parts of the compound word are recognized by the spelling dictionary, and the intent is to use two words.

Related rules

- CA1701: Resource string compound words should be cased correctly
- CA1709: Identifiers should be cased correctly
- CA1708: Identifiers should differ by more than case

See also

- Naming Guidelines
- Capitalization Conventions

CA1703: Resource strings should be spelled correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1703
Category	Microsoft.Naming
Breaking change	Non-breaking

Cause

A resource string contains one or more words that are not recognized by the Microsoft spelling checker library.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule parses the resource string into words (tokenizing compound words) and checks the spelling of each word/token. For information about the parsing algorithm, see CA1704: Identifiers should be spelled correctly.

How to fix violations

To fix a violation of this rule, use complete words that are correctly spelled or add the words to a custom dictionary. For information about how to use custom dictionaries, see CA1704: Identifiers should be spelled correctly.

Change the dictionary language

By default, the English (en) version of the spelling checker is used. If you want to change the language of the spelling checker, you can do so by adding one of the following attributes to your *AssemblyInfo.cs* or *AssemblyInfo.vb* file:

- Use AssemblyCultureAttribute to specify the culture if your resources are in a satellite assembly.
- Use NeutralResourcesLanguageAttribute to specify the *neutral culture* of your assembly if your resources are in the same assembly as your code.

> **IMPORTANT**
> If you set the culture to anything other than an English-based culture, this code analysis rule is silently disabled.

When to suppress warnings

Do not suppress a warning from this rule. Correctly spelled words reduce the time that is required to learn new software libraries.

Related rules

- CA1701: Resource string compound words should be cased correctly
- CA1704: Identifiers should be spelled correctly
- CA2204: Literals should be spelled correctly

CA1704: Identifiers should be spelled correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1704
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The name of an identifier contains one or more words that are not recognized by the Microsoft spelling checker library. This rule does not check constructors or special-named members such as get and set property accessors.

Rule description

This rule parses the identifier into tokens and checks the spelling of each token. The parsing algorithm performs the following transformations:

- Uppercase letters start a new token. For example, MyNameIsJoe tokenizes to "My", "Name", "Is", "Joe".
- For multiple uppercase letters, the last uppercase letter starts a new token. For example, GUIEditor tokenizes to "GUI", "Editor".
- Leading and trailing apostrophes are removed. For example, 'sender' tokenizes to "sender".
- Underscores signify the end of a token and are removed. For example, Hello_world tokenizes to "Hello", "world".
- Embedded ampersands are removed. For example, for&mat tokenizes to "format".

Language

The spell checker currently checks only against English-based culture dictionaries. You can change the culture of your project in the project file, by adding the **CodeAnalysisCulture** element.

For example:

```
<Project ...>
  <PropertyGroup>
    <CodeAnalysisCulture>en-AU</CodeAnalysisCulture>
```

> **IMPORTANT**
>
> If you set the culture to anything other than an English-based culture, this code analysis rule is silently disabled.

How to fix violations

To fix a violation of this rule, correct the spelling of the word or add the word to a custom dictionary.

To add words to a custom dictionary

Name the custom dictionary XML file *CustomDictionary.xml*. Place the dictionary in the installation directory of the tool, the project directory, or in the directory that is associated with the tool under the profile of the user (*%USERPROFILE%\Application Data\...*). To learn how to add the custom dictionary to a project in Visual Studio, see How to: Customize the Code Analysis Dictionary.

- Add words that should not cause a violation under the Dictionary/Words/Recognized path.
- Add words that should cause a violation under the Dictionary/Words/Unrecognized path.
- Add words that should be flagged as obsolete under the Dictionary/Words/Deprecated path. See the related rule topic CA1726: Use preferred terms for more information.
- Add exceptions to the acronym casing rules to the Dictionary/Acronyms/CasingExceptions path.

The following is an example of the structure of a custom dictionary file:

```
<Dictionary>
   <Words>
      <Unrecognized>
         <Word>cb</Word>
      </Unrecognized>
      <Recognized>
         <Word>stylesheet</Word>
         <Word>GotDotNet</Word>
      </Recognized>
      <Deprecated>
         <Term PreferredAlternate="EnterpriseServices">ComPlus</Term>
      </Deprecated>
   </Words>
   <Acronyms>
      <CasingExceptions>
         <Acronym>CJK</Acronym>
         <Acronym>Pi</Acronym>
      </CasingExceptions>
   </Acronyms>
</Dictionary>
```

When to suppress warnings

Suppress a warning from this rule only if the word is intentionally misspelled and the word applies to a limited set of the library. Correctly spelled words reduce the learning curve that is required for new software libraries.

Related rules

- CA2204: Literals should be spelled correctly
- CA1703: Resource strings should be spelled correctly
- CA1709: Identifiers should be cased correctly
- CA1708: Identifiers should differ by more than case
- CA1707: Identifiers should not contain underscores
- CA1726: Use preferred terms

See also

- How to: Customize the Code Analysis Dictionary

CA1707: Identifiers should not contain underscores

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1707
Category	Microsoft.Naming
Breaking change	Breaking - when raised on assemblies Non-breaking - when raised on type parameters

Cause

The name of an identifier contains the underscore (_) character.

Rule description

By convention, identifier names do not contain the underscore (_) character. The rule checks namespaces, types, members, and parameters.

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the learning curve that is required for new software libraries, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

Remove all underscore characters from the name.

When to suppress warnings

Do not suppress warnings for production code. However, it's safe to suppress this warning for test code. You can suppress warnings from this rule by setting its severity to **none**.

For well-known methods in Microsoft code that currently use an underscore and cannot be modified, the rule should be suppressed.

Related rules

- CA1709: Identifiers should be cased correctly
- CA1708: Identifiers should differ by more than case

CA1708: Identifiers should differ by more than case

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1708
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The names of two types, members, parameters, or fully qualified namespaces are identical when they're converted to lowercase.

By default, this rule only looks at externally visible types, members, and namespaces, but this is configurable.

Rule description

Identifiers for namespaces, types, members, and parameters cannot differ only by case because languages that target the common language runtime are not required to be case-sensitive. For example, Visual Basic is a widely used case-insensitive language.

This rule fires on publicly visible members only.

How to fix violations

Select a name that is unique when it is compared to other identifiers in a case-insensitive manner.

When to suppress warnings

Do not suppress a warning from this rule. The library might not be usable in all available languages in .NET.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1708.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Naming). For more information, see Configure FxCop analyzers.

Example of a violation

The following example demonstrates a violation of this rule.

```
using System;
namespace NamingLibrary
{
    public class Class1 // IdentifiersShouldDifferByMoreThanCase
    {
        protected string someProperty;

        public string SomeProperty
        {
            get { return someProperty; }
        }
    }
}
```

Related rules

- CA1709: Identifiers should be cased correctly

CA1709: Identifiers should be cased correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1709
Category	Microsoft.Naming
Breaking change	Breaking - when raised on assemblies, namespaces, types, members, and parameters. Non-breaking - when fired on generic type parameters.

Cause

The name of an identifier is not cased correctly.

- or -

The name of an identifier contains a two-letter acronym and the second letter is lowercase.

- or -

The name of an identifier contains an acronym of three or more uppercase letters.

Rule description

Naming conventions provide a common look for libraries that target the common language runtime. This consistency reduces the learning curve that's required for new software libraries, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

By convention, parameter names use camel casing, and namespace, type, and member names use Pascal casing. In a camel-cased name, the first letter is lowercase, and the first letter of any remaining words in the name is uppercase. Examples of camel-cased names are `packetSniffer`, `ioFile`, and `fatalErrorCode`. In a Pascal-cased name, the first letter is uppercase, and the first letter of any remaining words in the name is uppercase. Examples of Pascal-cased names are `PacketSniffer`, `IOFile`, and `FatalErrorCode`.

This rule splits the name into words based on the casing and checks any two-letter words against a list of common two-letter words, such as "In" or "My". If a match is not found, the word is assumed to be an acronym. In addition, this rule assumes it has found an acronym when the name contains either four uppercase letters in a row or three uppercase letters in a row at the end of the name.

By convention, two-letter acronyms use all uppercase letters, and acronyms of three or more characters use Pascal casing. The following examples use this naming convention: 'DB', 'CR', 'Cpa', and 'Ecma'. The following examples violate the convention: 'Io', 'XML', and 'DoD', and for non-parameter names, 'xp' and 'cpl'.

'ID' is special-cased to cause a violation of this rule. 'Id' is not an acronym but is an abbreviation for 'identification'.

How to fix violations

Change the name so that it is cased correctly.

When to suppress warnings

It is safe to suppress this warning if you have your own naming conventions, or if the identifier represents a proper name, for example, the name of a company or a technology.

You can also add specific terms, abbreviations, and acronyms that to a code analysis custom dictionary. Terms specified in the custom dictionary will not cause violations of this rule. For more information, see How to: Customize the Code Analysis Dictionary.

Related rules

CA1708: Identifiers should differ by more than case

CA1710: Identifiers should have correct suffix

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1710
Category	Microsoft.Naming
Breaking change	Breaking

Cause

An identifier does not have the correct suffix.

By default, this rule only looks at externally visible identifiers, but this is configurable.

Rule description

By convention, the names of types that extend certain base types or that implement certain interfaces, or types derived from these types, have a suffix that is associated with the base type or interface.

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the learning curve that is required for new software libraries, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

The following table lists the base types and interfaces that have associated suffixes.

BASE TYPE/INTERFACE	SUFFIX
System.Attribute	Attribute
System.EventArgs	EventArgs
System.Exception	Exception
System.Collections.ICollection	Collection
System.Collections.IDictionary	Dictionary
System.Collections.IEnumerable	Collection
System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>	Dictionary
System.Collections.Queue	Collection or Queue
System.Collections.Stack	Collection or Stack
System.Collections.Generic.ICollection<T>	Collection

BASE TYPE/INTERFACE	SUFFIX
System.Collections.Generic.IDictionary<TKey,TValue>	Dictionary
System.Data.DataSet	DataSet
System.Data.DataTable	Collection or DataTable
System.IO.Stream	Stream
System.Security.IPermission	Permission
System.Security.Policy.IMembershipCondition	Condition
An event-handler delegate.	EventHandler

Types that implement ICollection and are a generalized type of data structure, such as a dictionary, stack, or queue, are allowed names that provide meaningful information about the intended usage of the type.

Types that implement ICollection and are a collection of specific items have names that end with the word 'Collection'. For example, a collection of Queue objects would have the name 'QueueCollection'. The 'Collection' suffix signifies that the members of the collection can be enumerated by using the `foreach` (`For Each` in Visual Basic) statement.

Types that implement IDictionary or IReadOnlyDictionary<TKey,TValue> have names that end with the word 'Dictionary' even if the type also implements IEnumerable or ICollection. The 'Collection' and 'Dictionary' suffix naming conventions enable users to distinguish between the following two enumeration patterns.

Types with the 'Collection' suffix follow this enumeration pattern.

```
foreach(SomeType x in SomeCollection) { }
```

Types with the 'Dictionary' suffix follow this enumeration pattern.

```
foreach(SomeType x in SomeDictionary.Values) { }
```

A DataSet object consists of a collection of DataTable objects, which consist of collections of System.Data.DataColumn and System.Data.DataRow objects, among others. These collections implement ICollection through the base System.Data.InternalDataCollectionBase class.

How to fix violations

Rename the type so that it is suffixed with the correct term.

When to suppress warnings

It is safe to suppress a warning to use the 'Collection' suffix if the type is a generalized data structure that might be extended or that will hold an arbitrary set of diverse items. In this case, a name that provides meaningful information about the implementation, performance, or other characteristics of the data structure might make sense (for example, BinaryTree). In cases where the type represents a collection of a specific type (for example, StringCollection), do not suppress a warning from this rule because the suffix indicates that the type can be enumerated by using a `foreach` statement.

For other suffixes, do not suppress a warning from this rule. The suffix allows the intended usage to be evident from the type name.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1710.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Naming). For more information, see Configure FxCop analyzers.

Exclude indirect base types

You can configure whether to exclude indirect base types from the rule. By default, this option is set to true, which restricts analysis to the current base type.

```
dotnet_code_quality.CA1710.exclude_indirect_base_types = false
```

Related rules

CA1711: Identifiers should not have incorrect suffix

See also

- Attributes
- Handling and raising events

CA1711: Identifiers should not have incorrect suffix

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1711
Category	Microsoft.Naming
Breaking change	Breaking

Cause

An identifier has an incorrect suffix.

By default, this rule only looks at externally visible identifiers, but this is configurable.

Rule description

By convention, only the names of types that extend certain base types or that implement certain interfaces, or types derived from these types, should end with specific reserved suffixes. Other type names should not use these reserved suffixes.

The following table lists the reserved suffixes and the base types and interfaces with which they are associated.

SUFFIX	BASE TYPE/INTERFACE
Attribute	System.Attribute
Collection	System.Collections.ICollection System.Collections.IEnumerable System.Collections.Queue System.Collections.Stack System.Collections.Generic.ICollection<T> System.Data.DataSet System.Data.DataTable
Dictionary	System.Collections.IDictionary System.Collections.Generic.IDictionary<TKey,TValue>
EventArgs	System.EventArgs
EventHandler	An event-handler delegate
Exception	System.Exception

SUFFIX	BASE TYPE/INTERFACE
Flag	System.Enum
Flags	System.Enum
Permission	System.Security.IPermission
Queue	System.Collections.Queue
Stack	System.Collections.Stack
Stream	System.IO.Stream

In addition, the following suffixes should **not** be used:

- `Delegate`
- `Enum`
- `Impl` (use `Core` instead)
- `Ex` or similar suffix to distinguish it from an earlier version of the same type

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the learning curve that is required for new software libraries, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

Remove the suffix from the type name.

When to suppress warnings

Do not suppress a warning from this rule unless the suffix has an unambiguous meaning in the application domain.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

API surface

You can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1711.api_surface = private, internal
```

You can configure these options for just this rule, for all rules, or for all rules in this category (Naming). For more information, see Configure FxCop analyzers.

Allow suffixes

You can configure a list of allowed suffixes, with each suffix separated by the pipe character ("|"). For example, to specify that the rule should not run against Flag and Flags suffixes, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1711.allowed_suffixes = Flag|Flags
```

Related rules

- CA1710: Identifiers should have correct suffix

See also

- Attributes
- Handling and raising events

CA1712: Do not prefix enum values with type name

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1712
Category	Microsoft.Naming
Breaking change	Breaking

Cause

An enumeration contains a member whose name starts with the type name of the enumeration.

Rule description

Names of enumeration members are not prefixed with the type name because type information is expected to be provided by development tools.

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the time that is required for to learn a new software library, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

To fix a violation of this rule, remove the type name prefix from the enumeration member.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows an incorrectly named enumeration followed by the corrected version.

```
using System;

namespace NamingLibrary
{
   public enum DigitalImageMode
   {
      DigitalImageModeBitmap = 0,
      DigitalImageModeGrayscale = 1,
      DigitalImageModeIndexed = 2,
      DigitalImageModeRGB = 3
   }

   public enum DigitalImageMode2
   {
      Bitmap = 0,
      Grayscale = 1,
      Indexed = 2,
      RGB = 3
   }
}
```

```
using namespace System;

namespace NamingLibrary
{
   public enum class DigitalImageMode
   {
      DigitalImageModeBitmap = 0,
      DigitalImageModeGrayscale = 1,
      DigitalImageModeIndexed = 2,
      DigitalImageModeRGB = 3
   };

   public enum class DigitalImageMode2
   {
      Bitmap = 0,
      Grayscale = 1,
      Indexed = 2,
      RGB = 3
   };
}
```

```
Imports System

Namespace NamingLibrary

   Enum DigitalImageMode

      DigitalImageModeBitmap = 0
      DigitalImageModeGrayscale = 1
      DigitalImageModeIndexed = 2
      DigitalImageModeRGB = 3

   End Enum

   Enum DigitalImageMode2

      Bitmap = 0
      Grayscale = 1
      Indexed = 2
      RGB = 3

   End Enum

End Namespace
```

Related rules

CA1711: Identifiers should not have incorrect suffix

CA1027: Mark enums with FlagsAttribute

CA2217: Do not mark enums with FlagsAttribute

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

You can configure the number of enumeration values required to trigger the rule. For example, to specify that the rule is triggered if one or more the enum values starts with the enum type name, add the following key-value pair to an *.editorconfig* file in your project:

```
dotnet_code_quality.CA1712.enum_values_prefix_trigger = AnyEnumValue
```

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA1712.enum_values_prefix_trigger = AnyEnumValue`	The rule is triggered if *any* of the enum values starts with the enum type name.
`dotnet_code_quality.CA1712.enum_values_prefix_trigger = AllEnumValues`	The rule is triggered if *all* of the enum values start with the enum type name.
`dotnet_code_quality.CA1712.enum_values_prefix_trigger = Heuristic`	The rule is triggered using the default FxCop heuristic (that is, when at least 75% of the enum values start with the enum type name).

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Naming). For

more information, see Configure FxCop analyzers.

See also

- System.Enum

CA1713: Events should not have before or after prefix

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1713
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The name of an event starts with 'Before' or 'After'.

Rule description

Event names should describe the action that raises the event. To name related events that are raised in a specific sequence, use the present or past tense to indicate the relative position in the sequence of actions. For example, when naming a pair of events that is raised when closing a resource, you might name it 'Closing' and 'Closed', instead of 'BeforeClose' and 'AfterClose'.

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the learning curve that is required for new software libraries, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

Remove the prefix from the event name, and consider changing the name to use the present or past tense of a verb.

When to suppress warnings

Do not suppress a warning from this rule.

CA1714: Flags enums should have plural names

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1714
Category	Microsoft.Naming
Breaking change	Breaking

Cause

An enumeration has the System.FlagsAttribute and its name does not end in 's'.

By default, this rule only looks at externally visible enumerations, but this is configurable.

Rule description

Types that are marked with FlagsAttribute have names that are plural because the attribute indicates that more than one value can be specified. For example, an enumeration that defines the days of the week might be intended for use in an application where you can specify multiple days. This enumeration should have the FlagsAttribute and could be called 'Days'. A similar enumeration that allows only a single day to be specified would not have the attribute, and could be called 'Day'.

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the learning curve that is required for new software libraries, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

Make the name of the enumeration a plural word, or remove the FlagsAttribute attribute if multiple enumeration values should not be specified simultaneously.

When to suppress warnings

It is safe to suppress a violation if the name is a plural word but does not end in 's'. For example, if the multiple-day enumeration that was described previously were named 'DaysOfTheWeek', this would violate the logic of the rule but not its intent. Such violations should be suppressed.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1714.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Naming). For more

information, see Configure FxCop analyzers.

Related rules

- CA1027: Mark enums with FlagsAttribute
- CA2217: Do not mark enums with FlagsAttribute

See also

- System.FlagsAttribute
- Enum design

CA1715: Identifiers should have correct prefix

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1715
Category	Microsoft.Naming
Breaking change	Breaking - when fired on interfaces. Non-breaking - when raised on generic type parameters.

Cause

The name of an interface does not start with an uppercase 'I'.

-or-

The name of a generic type parameter on a type or method does not start with an uppercase 'T'.

By default, this rule only looks at externally visible interfaces, types, and methods, but this is configurable.

Rule description

By convention, the names of certain programming elements start with a specific prefix.

Interface names should start with an uppercase 'I' followed by another uppercase letter. This rule reports violations for interface names such as 'MyInterface' and 'IsolatedInterface'.

Generic type parameter names should start with an uppercase 'T' and optionally may be followed by another uppercase letter. This rule reports violations for generic type parameter names such as 'V' and 'Type'.

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the learning curve that is required for new software libraries, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your code this rule analyzes. For more information, see Configure FxCop analyzers.

Single-character type parameters

You can configure whether or not to exclude single-character type parameters from this rule. For example, to specify that this rule *should not* analyze single-character type parameters, add one of the following key-value pairs to an .editorconfig file in your project:

```
# Package version 2.9.0 and later
dotnet_code_quality.CA1715.exclude_single_letter_type_parameters = true

# Package version 2.6.3 and earlier
dotnet_code_quality.CA2007.allow_single_letter_type_parameters = true
```

> **NOTE**
> This rule never fires for a type parameter named `T`, for example, `Collection<T>`.

API surface

You can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1715.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Naming).

How to fix violations

Rename the identifier so that it is correctly prefixed.

When to suppress warnings

Do not suppress a warning from this rule.

Interface naming example

The following code snippet shows an incorrectly named interface:

```
using namespace System;

namespace Samples
{
    public interface class Book     // Violates this rule
    {
        property String^ Title
        {
            String^ get();
        }
        void Read();
    };
}
```

```
Imports System

Namespace Samples

    Public Interface Book      ' Violates this rule

        ReadOnly Property Title() As String

        Sub Read()

    End Interface

End Namespace
```

```
using System;

namespace Samples
{
    public interface Book   // Violates this rule
    {
        string Title
        {
            get;
        }

        void Read();
    }
}
```

The following code snippet fixes the previous violation by prefixing the interface with 'I':

```
using System;

namespace Samples
{
    public interface IBook      // Fixes the violation by prefixing the interface with 'I'
    {
        string Title
        {
            get;
        }

        void Read();
    }
}
```

```
using namespace System;

namespace Samples
{
    public interface class IBook  // Fixes the violation by prefixing the interface with 'I'
    {
        property String^ Title
        {
            String^ get();
        }
        void Read();
    };
}
```

```
Imports System

Namespace Samples

    Public Interface IBook  ' Fixes the violation by prefixing the interface with 'I'

        ReadOnly Property Title() As String

        Sub Read()

    End Interface

End Namespace
```

Type parameter naming example

The following code snippet shows an incorrectly named generic type parameter:

```
using namespace System;

namespace Samples
{
    generic <typename Item>     // Violates this rule
    public ref class Collection
    {

    };
}
```

```
Imports System

Namespace Samples

    Public Class Collection(Of Item)    ' Violates this rule

    End Class

End Namespace
```

```
using System;

namespace Samples
{
    public class Collection<Item>   // Violates this rule
    {

    }
}
```

The following code snippet fixes the previous violation by prefixing the generic type parameter with 'T':

```
using namespace System;

namespace Samples
{
    generic <typename TItem>  // Fixes the violation by prefixing the generic type parameter with 'T'
    public ref class Collection
    {

    };
}
```

```
using System;

namespace Samples
{
    public class Collection<TItem>  // Fixes the violation by prefixing the generic type parameter with 'T'

    {

    }
}
```

```
Imports System

Namespace Samples

    Public Class Collection(Of TItem)  ' Fixes the violation by prefixing the generic type parameter with 'T'

    End Class

End Namespace
```

Related rules

- CA1722: Identifiers should not have incorrect prefix

CA1716: Identifiers should not match keywords

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1716
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The name of a namespace, type, or virtual or interface member matches a reserved keyword in a programming language.

By default, this rule only looks at externally visible namespaces, types, and members, but you can configure visibility and symbol kinds.

Rule description

Identifiers for namespaces, types, and virtual and interface members should not match keywords that are defined by languages that target the common language runtime. Depending on the language that is used and the keyword, compiler errors and ambiguities can make the library difficult to use.

This rule checks against keywords in the following languages:

- Visual Basic
- C#
- C++/CLI

Case-insensitive comparison is used for Visual Basic keywords, and case-sensitive comparison is used for the other languages.

How to fix violations

Select a name that does not appear in the list of keywords.

When to suppress warnings

You can suppress a warning from this rule if you're convinced that the identifier won't confuse users of the API, and that the library is usable in all available languages in .NET.

Configurability

Api surface

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1716.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Naming). For more information, see Configure FxCop analyzers.

Analyzed symbol kinds

You can configure the members types that will be analyzed by this rule. Use one or more of these values `Namespace`, `NamedType`, `Method`, `Property`, `Event`, `Parameter` as a comma separated list. Note that the default value includes all of the above members.

```
dotnet_code_quality.CA1716.analyzed_symbol_kinds = Namespace, NamedType, Method, Property, Event
```

CA1717: Only FlagsAttribute enums should have plural names

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1717
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The name of an enumeration ends in a plural word and the enumeration is not marked with the System.FlagsAttribute attribute.

By default, this rule only looks at externally visible enumerations, but this is configurable.

Rule description

Naming conventions dictate that a plural name for an enumeration indicates that more than one value of the enumeration can be specified simultaneously. The FlagsAttribute tells compilers that the enumeration should be treated as a bit field that enables bitwise operations on the enumeration.

If only one value of an enumeration can be specified at a time, the name of the enumeration should be a singular word. For example, an enumeration that defines the days of the week might be intended for use in an application where you can specify multiple days. This enumeration should have the FlagsAttribute and could be called 'Days'. A similar enumeration that allows only a single day to be specified would not have the attribute, and could be called 'Day'.

Naming conventions provide a common look for libraries that target the common language runtime. This reduces the time that is required to learn a new software library, and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

Make the name of the enumeration a singular word or add the FlagsAttribute.

When to suppress warnings

It is safe to suppress a warning from the rule if the name ends in a singular word.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1717.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Naming). For more information, see Configure FxCop analyzers.

Related rules

- CA1714: Flags enums should have plural names
- CA1027: Mark enums with FlagsAttribute
- CA2217: Do not mark enums with FlagsAttribute

See also

- System.FlagsAttribute
- Enum design

CA1719: Parameter names should not match member names

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1719
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The name of an externally visible member matches, in a case-insensitive comparison, the name of one of its parameters.

Rule description

A parameter name should communicate the meaning of a parameter and a member name should communicate the meaning of a member. It would be a rare design where these were the same. Naming a parameter the same as its member name is unintuitive and makes the library difficult to use.

How to fix violations

Select a parameter name that does not match the member name.

When to suppress warnings

For new development, no known scenarios occur where you must suppress a warning from this rule. For shipping libraries, you might have to suppress a warning from this rule.

Related rules

CA1709: Identifiers should be cased correctly

CA1708: Identifiers should differ by more than case

CA1707: Identifiers should not contain underscores

CA1720: Identifiers should not contain type names

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1720
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The name of a parameter in a member contains a data type name.

-or-

The name of a member contains a language-specific data type name.

By default, this rule only looks at externally visible members, but this is configurable.

Rule description

Names of parameters and members are better used to communicate their meaning than to describe their type, which is expected to be provided by development tools. For names of members, if a data type name must be used, use a language-independent name instead of a language-specific one. For example, instead of the C# type name `int`, use the language-independent data type name, `Int32`.

Each discrete token in the name of the parameter or member is checked against the following language-specific data type names in a case-insensitive manner:

- Bool
- WChar
- Int8
- UInt8
- Short
- UShort
- Int
- UInt
- Integer
- UInteger
- Long
- ULong
- Unsigned
- Signed
- Float
- Float32
- Float64

In addition, the names of a parameter are also checked against the following language-independent data type names in a case-insensitive manner:

- Object
- Boolean
- Char
- String
- SByte
- Byte
- UByte
- Int16
- UInt16
- Int32
- UInt32
- Int64
- UInt64
- IntPtr
- Ptr
- Pointer
- UInptr
- UPtr
- UPointer
- Single
- Double
- Decimal
- Guid

How to fix violations

If fired against a parameter:

Replace the data type identifier in the name of the parameter with either a term that better describes its meaning or a more generic term, such as 'value'.

If fired against a member:

Replace the language-specific data type identifier in the name of the member with a term that better describes its meaning, a language-independent equivalent, or a more generic term, such as 'value'.

When to suppress warnings

Occasional use of type-based parameter and member names might be appropriate. However, for new development, no known scenarios occur where you should suppress a warning from this rule. For libraries that have previously shipped, you might have to suppress a warning from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1720.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Naming). For more information, see Configure FxCop analyzers.

Related rules

- CA1709: Identifiers should be cased correctly
- CA1708: Identifiers should differ by more than case
- CA1707: Identifiers should not contain underscores
- CA1719: Parameter names should not match member names

CA1721: Property names should not match get methods

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1721
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The name of a member starts with 'Get' and otherwise matches the name of a property. For example, a type that contains a method that is named 'GetColor' and a property that is named 'Color' cause a rule violation. This rule will not fire if either the property or the method is marked with the ObsoleteAttribute.

By default, this rule only looks at externally visible members and properties, but this is configurable.

Rule description

"Get" methods and properties should have names that clearly distinguish their function.

Naming conventions provide a common look for libraries that target the common language runtime. This consistency reduces the time that's required to learn a new software library and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

Change the name so that it does not match the name of a method that is prefixed with 'Get'.

When to suppress warnings

Do not suppress a warning from this rule.

> **NOTE**
>
> This warning may be excluded if the "Get" method is caused by implementing IExtenderProvider interface.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1721.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Naming). For more information, see Configure FxCop analyzers.

Example

The following example contains a method and property that violate this rule.

```
using System;

namespace NamingLibrary
{
    public class Test
    {
        public DateTime Date
        {
            get { return DateTime.Today; }
        }
         // Violates rule: PropertyNamesShouldNotMatchGetMethods.
        public string GetDate()
        {
            return this.Date.ToString();
        }
    }
}
```

```
Imports System

Namespace NamingLibrary

Public Class Test

    Public ReadOnly Property [Date]() As DateTime
        Get
            Return DateTime.Today
        End Get
    End Property

     ' Violates rule: PropertyNamesShouldNotMatchGetMethods.
    Public Function GetDate() As String
        Return Me.Date.ToString()
    End Function

End Class

End Namespace
```

Related rules

- CA1024: Use properties where appropriate

CA1722: Identifiers should not have incorrect prefix

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1722
Category	Microsoft.Naming
Breaking change	Breaking

Cause

An identifier has an incorrect prefix.

Rule description

By convention, only certain programming elements have names that begin with a specific prefix.

Type names do not have a specific prefix and should not be prefixed with a 'C'. This rule reports violations for type names such as 'CMyClass' and does not report violations for type names such as 'Cache'.

Naming conventions provide a common look for libraries that target the common language runtime. This consistency reduces the learning curve that's required for new software libraries and increases customer confidence that the library was developed by someone who has expertise in developing managed code.

How to fix violations

Remove the prefix from the identifier.

When to suppress warnings

Do not suppress a warning from this rule.

Related rules

CA1715: Identifiers should have correct prefix

CA1725: Parameter names should match base declaration

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1725
Category	Microsoft.Naming
Breaking change	Breaking

Cause

The name of a parameter in a method override does not match the name of the parameter in the base declaration of the method or the name of the parameter in the interface declaration of the method.

By default, this rule only looks at externally visible methods, but this is configurable.

Rule description

Consistent naming of parameters in an override hierarchy increases the usability of the method overrides. A parameter name in a derived method that differs from the name in the base declaration can cause confusion about whether the method is an override of the base method or a new overload of the method.

How to fix violations

To fix a violation of this rule, rename the parameter to match the base declaration. The fix is a breaking change for COM visible methods.

When to suppress warnings

Do not suppress a warning from this rule except for COM visible methods in libraries that have previously shipped.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1725.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Naming). For more information, see Configure FxCop analyzers.

CA1724: Type names should not match namespaces

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1724
Category	Microsoft.Naming
Breaking change	Breaking

Cause

A type name matches a referenced namespace name that has one or more externally visible types. The name comparison is case-insensitive.

Rule description

User-created type names should not match the names of referenced namespaces that have externally visible types. Violating this rule can reduce the usability of your library.

How to fix violations

Rename the type such that it doesn't match the name of a referenced namespace that has externally visible types.

When to suppress warnings

For new development, no known scenarios occur where you must suppress a warning from this rule. Before you suppress the warning, carefully consider how the users of your library might be confused by the matching name. For shipping libraries, you might have to suppress a warning from this rule.

Example

```
namespace MyNamespace
{
    // This class violates the rule
    public class System
    {
    }
}
```

CA1726: Use preferred terms

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1726
Category	Microsoft.Naming
Breaking change	Breaking - when fired on assemblies Non-breaking - when fired on type parameters

Cause

The name of an externally visible identifier includes a term for which an alternative, preferred term exists. Or, the name includes the term Flag or Flags.

Rule description

This rule parses an identifier into tokens. Each single token and each contiguous dual token combination is compared to terms that are built into the rule and in the Deprecated section of any custom dictionaries. The following table shows the terms that are built into the rule and their preferred alternatives.

OBSOLETE TERM	PREFERRED TERM
`Arent`	`AreNot`
`Cancelled`	`Canceled`
`Cant`	`Cannot`
`ComPlus`	`EnterpriseServices`
`Couldnt`	`CouldNot`
`Didnt`	`DidNot`
`Doesnt`	`DoesNot`
`Dont`	`DoNot`
`Flag` or `Flags`	There is no replacement term. Do not use.
`Hadnt`	`HadNot`
`Hasnt`	`HasNot`

OBSOLETE TERM	PREFERRED TERM
`Havent`	`HaveNot`
`Indices`	`Indexes`
`Isnt`	`IsNot`
`LogIn`	`LogOn`
`LogOut`	`LogOff`
`Shouldnt`	`ShouldNot`
`SignOn`	`SignIn`
`SignOff`	`SignOut`
`Wasnt`	`WasNot`
`Werent`	`WereNot`
`Wont`	`WillNot`
`Wouldnt`	`WouldNot`
`Writeable`	`Writable`

How to fix violations

To fix a violation of this rule, replace the term with the preferred alternative term.

When to suppress warnings

Suppress a warning from this rule only if the name of the identifier is intentional and relates specifically to the original term instead of the preferred term.

Related rules

Naming Warnings

Performance Warnings

8/4/2020 • 5 minutes to read • Edit Online

Performance warnings support high-performance libraries and applications.

In This Section

RULE	DESCRIPTION
CA1800: Do not cast unnecessarily	Duplicate casts decrease performance, especially when the casts are performed in compact iteration statements.
CA1801: Review unused parameters	A method signature includes a parameter that is not used in the method body.
CA1802: Use Literals Where Appropriate	A field is declared static and read-only (Shared and ReadOnly in Visual Basic), and is initialized with a value that is computable at compile time. Because the value that is assigned to the targeted field is computable at compile time, change the declaration to a const (Const in Visual Basic) field so that the value is computed at compile time instead of at run time.
CA1804: Remove unused locals	Unused local variables and unnecessary assignments increase the size of an assembly and decrease performance.
CA1805: Do not initialize unnecessarily	The .NET runtime initializes all fields of reference types to their default values before running the constructor. In most cases, explicitly initializing a field to its default value is redundant, which adds to maintenance costs and may degrade performance (such as with increased assembly size).
CA1806: Do not ignore method results	A new object is created but never used, or a method that creates and returns a new string is called and the new string is never used, or a Component Object Model (COM) or P/Invoke method returns an HRESULT or error code that is never used.
CA1809: Avoid excessive locals	A common performance optimization is to store a value in a processor register instead of memory, which is referred to as "enregistering the value". To increase the chance that all local variables are enregistered, limit the number of local variables to 64.
CA1810: Initialize reference type static fields inline	When a type declares an explicit static constructor, the just-in-time (JIT) compiler adds a check to each static method and instance constructor of the type to make sure that the static constructor was previously called. Static constructor checks can decrease performance.
CA1811: Avoid uncalled private code	A private or internal (assembly-level) member does not have callers in the assembly, it is not invoked by the common language runtime, and it is not invoked by a delegate.

RULE	DESCRIPTION
CA1812: Avoid uninstantiated internal classes	An instance of an assembly-level type is not created by code in the assembly.
CA1813: Avoid unsealed attributes	.NET provides methods for retrieving custom attributes. By default, these methods search the attribute inheritance hierarchy. Sealing the attribute eliminates the search through the inheritance hierarchy and can improve performance.
CA1814: Prefer jagged arrays over multidimensional	A jagged array is an array whose elements are arrays. The arrays that make up the elements can be of different sizes, which can result in less wasted space for some sets of data.
CA1815: Override equals and operator equals on value types	For value types, the inherited implementation of Equals uses the Reflection library and compares the contents of all fields. Reflection is computationally expensive, and comparing every field for equality might be unnecessary. If you expect users to compare or sort instances, or to use instances as hash table keys, your value type should implement Equals.
CA1816: Call GC.SuppressFinalize correctly	A method that is an implementation of Dispose does not call GC.SuppressFinalize, or a method that is not an implementation of Dispose calls GC.SuppressFinalize, or a method calls GC.SuppressFinalize and passes something other than this (Me in Visual Basic).
CA1819: Properties should not return arrays	Arrays that are returned by properties are not write-protected, even if the property is read-only. To keep the array tamper-proof, the property must return a copy of the array. Typically, users will not understand the adverse performance implications of calling such a property.
CA1820: Test for empty strings using string length	Comparing strings by using the String.Length property or the String.IsNullOrEmpty method is significantly faster than using Equals.
CA1821: Remove empty finalizers	Whenever you can, avoid finalizers because of the additional performance overhead that is involved in tracking object lifetime. An empty finalizer incurs added overhead without any benefit.
CA1822: Mark members as static	Members that do not access instance data or call instance methods can be marked as static (Shared in Visual Basic). After you mark the methods as static, the compiler will emit nonvirtual call sites to these members. This can give you a measurable performance gain for performance-sensitive code.
CA1823: Avoid unused private fields	Private fields were detected that do not appear to be accessed in the assembly.
CA1824: Mark assemblies with NeutralResourcesLanguageAttribute	The NeutralResourcesLanguage attribute informs the ResourceManager of the language that was used to display the resources of a neutral culture for an assembly. This improves lookup performance for the first resource that you load and can reduce your working set.

RULE	DESCRIPTION
CA1825: Avoid zero-length array allocations	Initializing a zero-length array leads to unnecessary memory allocation. Instead, use the statically allocated empty array instance by calling Array.Empty. The memory allocation is shared across all invocations of this method.
CA1826: Use property instead of Linq Enumerable method	Enumerable LINQ method was used on a type that supports an equivalent, more efficient property.
CA1827: Do not use Count/LongCount when Any can be used	Count or LongCount method was used where Any method would be more efficient.
CA1828: Do not use CountAsync/LongCountAsync when AnyAsync can be used	CountAsync or LongCountAsync method was used where AnyAsync method would be more efficient.
CA1829: Use Length/Count property instead of Enumerable.Count method	Count LINQ method was used on a type that supports an equivalent, more efficient `Length` or `Count` property.
CA1830: Prefer strongly-typed Append and Insert method overloads on StringBuilder	Append and Insert provide overloads for multiple types beyond System.String. When possible, prefer the strongly-typed overloads over using ToString() and the string-based overload.
CA1831: Use AsSpan instead of Range-based indexers for string when appropriate	When using a range-indexer on a string and implicitly assigning the value to a ReadOnlySpan<char> type, the method Substring will be used instead of Slice, which produces a copy of requested portion of the string.
CA1832: Use AsSpan or AsMemory instead of Range-based indexers for getting ReadOnlySpan or ReadOnlyMemory portion of an array	When using a range-indexer on an array and implicitly assigning the value to a ReadOnlySpan<T> or ReadOnlyMemory<T> type, the method GetSubArray will be used instead of Slice, which produces a copy of requested portion of the array.
CA1833: Use AsSpan or AsMemory instead of Range-based indexers for getting Span or Memory portion of an array	When using a range-indexer on an array and implicitly assigning the value to a Span<T> or Memory<T> type, the method GetSubArray will be used instead of Slice, which produces a copy of requested portion of the array.
CA1835: Prefer the 'Memory'-based overloads for 'ReadAsync' and 'WriteAsync'	'Stream' has a 'ReadAsync' overload that takes a 'Memory<Byte>' as the first argument, and a 'WriteAsync' overload that takes a 'ReadOnlyMemory<Byte>' as the first argument. Prefer calling the memory based overloads, which are more efficient.
CA1836: Prefer `IsEmpty` over `Count` when available	Prefer `IsEmpty` property that is more efficient than `Count`, `Length`, Count<TSource>(IEnumerable<TSource>) or LongCount<TSource>(IEnumerable<TSource>) to determine whether the object contains or not any items.

CA1800: Do not cast unnecessarily

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1800
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A method performs duplicate casts on one of its arguments or local variables.

For complete analysis by this rule, the tested assembly must be built by using debugging information, and the associated program database (.pdb) file must be available.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Duplicate casts decrease performance, especially when the casts are performed in compact iteration statements. For explicit duplicate cast operations, store the result of the cast in a local variable and use the local variable instead of the duplicate cast operations.

If the C# `is` operator is used to test whether the cast will succeed before the actual cast is performed, consider testing the result of the `as` operator instead. This provides the same functionality without the implicit cast operation that is performed by the `is` operator. Or, in C# 7.0 and later, use the `is` operator with pattern matching to check the type conversion and cast the expression to a variable of that type in one step.

How to fix violations

To fix a violation of this rule, modify the method implementation to minimize the number of cast operations.

When to suppress warnings

It is safe to suppress a warning from this rule, or to ignore the rule completely, if performance is not a concern.

Examples

The following example shows a method that violates the rule by using the C# `is` operator. A second method satisfies the rule by replacing the `is` operator with a test against the result of the `as` operator, which decreases the number of cast operations per iteration from two to one. A third method also satisfies the rule by using `is` with pattern matching to create a variable of the desired type if the type conversion would succeed.

```
using System;
using System.Collections;
using System.Windows.Forms;

namespace PerformanceLibrary
{
   public sealed class SomeClass
   {
      private SomeClass() {}

      // This method violates the rule.
      public static void UnderPerforming(ArrayList list)
      {
         foreach(object obj in list)
         {
            // The 'is' statement performs a cast operation.
            if(obj is Control)
            {
               // The 'as' statement performs a duplicate cast operation.
               Control aControl = obj as Control;
               // Use aControl.
            }

         }
      }

      // This method satisfies the rule by checking
      // the result of the as operation.
      public static void BetterPerforming1(ArrayList list)
      {
          foreach (object obj in list)
          {
              Control aControl = obj as Control;
              if (aControl != null)
              {
                  // Use aControl.
                  Console.WriteLine(aControl.Name);
              }
          }
      }

      // This method also satisfies the rule by using
      // the is operator with a type pattern (C# 7.0).
      public static void BetterPerforming2(ArrayList list)
      {
          foreach (object obj in list)
          {
              if (obj is Control aControl)
              {
                  // Use aControl.
                  Console.WriteLine(aControl.Name);
              }
          }
      }
   }
}
```

The following example shows a method, `start_Click`, that has multiple duplicate explicit casts, which violates the rule, and a method, `reset_Click`, which satisfies the rule by storing the cast in a local variable.

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Namespace PerformanceLibrary

   Public Class SomeForm : Inherits Form

      Dim start, reset As Button

      Sub New()

         start = New Button()
         reset = New Button()
         AddHandler start.Click, AddressOf start_Click
         AddHandler reset.Click, AddressOf reset_Click
         Controls.Add(start)
         Controls.Add(reset)

      End Sub

      ' This method violates the rule.
      Private Sub start_Click(sender As Object, e As EventArgs)

         Dim controlSize As Size = DirectCast(sender, Control).Size
         Dim rightToLeftValue As RightToLeft = _
            DirectCast(sender, Control).RightToLeft
         Dim parent As Control = DirectCast(sender, Control)

      End Sub

      ' This method satisfies the rule.
      Private Sub reset_Click(sender As Object, e As EventArgs)

         Dim someControl As Control = DirectCast(sender, Control)
         Dim controlSize As Size = someControl.Size
         Dim rightToLeftValue As RightToLeft = someControl.RightToLeft
         Dim parent As Control = someControl

      End Sub

   End Class

End Namespace
```

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace PerformanceLibrary
{
   public class SomeForm : Form
   {
      Button start, reset;

      public SomeForm()
      {
         start = new Button();
         reset = new Button();
         start.Click += new EventHandler(start_Click);
         reset.Click += new EventHandler(reset_Click);
         Controls.Add(start);
         Controls.Add(reset);
      }

      // This method violates the rule.
      void start_Click(object sender, EventArgs e)
      {
         Size controlSize = ((Control)sender).Size;
         RightToLeft rightToLeftValue = ((Control)sender).RightToLeft;
         Control parent = (Control)sender;
      }

      // This method satisfies the rule.
      void reset_Click(object sender, EventArgs e)
      {
         Control someControl = (Control)sender;
         Size controlSize = someControl.Size;
         RightToLeft rightToLeftValue = someControl.RightToLeft;
         Control parent = someControl;
      }
   }
}
```

See also

- as (C# reference)
- is (C# reference)

CA1801: Review unused parameters

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1801
Category	Microsoft.Usage
Breaking change	Non-breaking - If the member is not visible outside the assembly, regardless of the change you make. Non-breaking - If you change the member to use the parameter within its body. Breaking - If you remove the parameter and it is visible outside the assembly.

Cause

A method signature includes a parameter that's not used in the method body.

This rule does not examine the following kinds of methods:

- Methods referenced by a delegate.
- Methods used as event handlers.
- Serialization constructors (see guidelines).
- Serialization GetObjectData method.
- Methods declared with the `abstract` (`MustOverride` in Visual Basic) modifier.
- Methods declared with the `virtual` (`Overridable` in Visual Basic) modifier.
- Methods declared with the `override` (`Overrides` in Visual Basic) modifier.
- Methods declared with the `extern` (`Declare` statement in Visual Basic) modifier.

If you're using FxCop analyzers, this rule does not flag parameters that are named with the discard symbol, for example, `_`, `_1`, and `_2`. This reduces warning noise on parameters that are needed for signature requirements, for example, a method used as a delegate, a parameter with special attributes, or a parameter whose value is implicitly accessed at run time by a framework but is not referenced in code.

Rule description

Review parameters in non-virtual methods that are not used in the method body to make sure no incorrectness exists around failure to access them. Unused parameters incur maintenance and performance costs.

Sometimes, a violation of this rule can point to an implementation bug in the method. For example, the parameter should have been used in the method body. Suppress warnings of this rule if the parameter must exist because of backward compatibility.

How to fix violations

To fix a violation of this rule, remove the unused parameter (a breaking change), or use the parameter in the method body (a non-breaking change).

When to suppress warnings

It is safe to suppress a warning from this rule:

- In previously shipped code for which the fix would be a breaking change.
- For the `this` parameter in a custom extension method for Microsoft.VisualStudio.TestTools.UnitTesting.Assert. The functions in the Assert class are static, so there's no need to access the `this` parameter in the method body.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1801.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Performance). For more information, see Configure FxCop analyzers.

Example

The following example shows two methods. One method violates the rule and the other method satisfies the rule.

```
using System;
using System.Globalization;

namespace Samples
{
    public static class TestClass
    {
        // This method violates the rule.
        public static string GetSomething(int first, int second)
        {
            return first.ToString(CultureInfo.InvariantCulture);
        }

        // This method satisfies the rule.
        public static string GetSomethingElse(int first)
        {
            return first.ToString(CultureInfo.InvariantCulture);
        }
    }
}
```

Related rules

CA1811: Avoid uncalled private code

CA1812: Avoid uninstantiated internal classes

CA1804: Remove unused locals

CA2229: Implement serialization constructors

CA1802: Use Literals Where Appropriate

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1802
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A field is declared `static` and `readonly` (`Shared` and `ReadOnly` in Visual Basic), and is initialized with a value that is computable at compile time.

By default, this rule only looks at externally visible, static, readonly fields, but this is configurable.

Rule description

The value of a `static readonly` field is computed at run time when the static constructor for the declaring type is called. If the `static readonly` field is initialized when it is declared and a static constructor is not declared explicitly, the compiler emits a static constructor to initialize the field.

The value of a `const` field is computed at compile time and stored in the metadata, which increases run-time performance when it is compared to a `static readonly` field.

Because the value assigned to the targeted field is computable at compile time, change the declaration to a `const` field so that the value is computed at compile time instead of at run time.

How to fix violations

To fix a violation of this rule, replace the `static` and `readonly` modifiers with the `const` modifier.

> **NOTE**
>
> The use of the (const modifier is not recommended for all scenarios.

When to suppress warnings

It is safe to suppress a warning from this rule, or disable the rule, if performance is not of concern.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Api Surface

You can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1802.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Performance). For more information, see Configure FxCop analyzers.

Required modifiers

You can configure this rule to override the required field modifiers. By default, `static` and `readonly` are both required modifiers for fields that are analyzed. You can override this to a comma separated listed of one or more modifier values from the below table:

OPTION VALUE	SUMMARY
`none`	No modifier requirement.
`static` or `Shared`	Must be declared as 'static' ('Shared' in Visual Basic).
`const`	Must be declared as 'const'.
`readonly`	Must be declared as 'readonly'.

For example, to specify that the rule should run against both static and instance fields, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA1802.required_modifiers = none
```

Example

The following example shows a type, `UseReadOnly`, that violates the rule and a type, `UseConstant`, that satisfies the rule.

```
Imports System

Namespace PerformanceLibrary

   ' This class violates the rule.
   Public Class UseReadOnly

      Shared ReadOnly x As Integer = 3
      Shared ReadOnly y As Double = x + 2.1
      Shared ReadOnly s As String = "readonly"

   End Class

   ' This class satisfies the rule.
   Public Class UseConstant

      Const x As Integer = 3
      Const y As Double = x + 2.1
      Const s As String = "const"

   End Class

End Namespace
```

```
using System;

namespace PerformanceLibrary
{
   // This class violates the rule.
   public class UseReadOnly
   {
      static readonly int x = 3;
      static readonly double y = x + 2.1;
      static readonly string s = "readonly";
   }

   // This class satisfies the rule.
   public class UseConstant
   {
      const int x = 3;
      const double y = x + 2.1;
      const string s = "const";
   }
}
```

CA1804: Remove unused locals

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1804
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A method declares a local variable but does not use the variable except possibly as the recipient of an assignment statement. For analysis by this rule, the tested assembly must be built with debugging information and the associated program database (.pdb) file must be available.

Rule description

Unused local variables and unnecessary assignments increase the size of an assembly and decrease performance.

How to fix violations

To fix a violation of this rule, remove or use the local variable.

> **NOTE**
>
> The C# compiler removes unused local variables when the `optimize` option is enabled.

When to suppress warnings

Suppress a warning from this rule if the variable was compiler emitted. It is also safe to suppress a warning from this rule, or to disable the rule, if performance and code maintenance are not primary concerns.

Example

The following example shows several unused local variables.

```
Imports System
Imports System.Windows.Forms

Namespace PerformanceLibrary

   Public Class UnusedLocals

      Sub SomeMethod()

         Dim unusedInteger As Integer
         Dim unusedString As String = "hello"
         Dim unusedArray As String() = Environment.GetLogicalDrives()
         Dim unusedButton As New Button()

      End Sub

   End Class

End Namespace
```

```
using System;
using System.Windows.Forms;

namespace PerformanceLibrary
{
   public class UnusedLocals
   {
      public void SomeMethod()
      {
         int unusedInteger;
         string unusedString = "hello";
         string[] unusedArray = Environment.GetLogicalDrives();
         Button unusedButton = new Button();
      }
   }
}
```

Related rules

CA1809: Avoid excessive locals

CA1811: Avoid uncalled private code

CA1812: Avoid uninstantiated internal classes

CA1801: Review unused parameters

CA1805: Do not initialize unnecessarily.

8/4/2020 • 2 minutes to read • Edit Online

CheckId	CA1805
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A field of a class is explicitly initialized to the default value of that field's type.

Rule description

The .NET runtime initializes all fields of reference types to their default values before running the constructor. In most cases, explicitly initializing a field to its default value in a constructor is redundant, adding maintenance costs and potentially degrading performance (such as with increased assembly size), and the explicit initialization can be removed.

How to fix violations

In most cases, the proper fix is to delete the unnecessary initialization.

```
class C
{
    // Violation
    int _value1 = 0;

    // Fixed
    int _value1;
}
```

In some cases, deleting the initialization may result in subsequent CS0649 warnings being issued due to the field retaining its default value forever. In such cases, a better fix may be to delete the field entirely or replace it with a property:

```
class C
{
    // Violation
    private static readonly int s_value = 0;

    // Fixed
    private static int Value => 0;
}
```

When to suppress warnings

It is always safe to suppress the warning, as the warning simply highlights potentially unnecessary code and work

that may be avoided.

See also

- Performance warnings

CA1806: Do not ignore method results

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1806
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

There are several possible reasons for this warning:

- A new object is created but never used.
- A method that creates and returns a new string is called and the new string is never used.
- A COM or P/Invoke method that returns a HRESULT or error code that is never used.
- A LINQ method that returns a result that is never used.

Rule description

Unnecessary object creation and the associated garbage collection of the unused object degrade performance.

Strings are immutable and methods such as String.ToUpper returns a new instance of a string instead of modifying the instance of the string in the calling method.

Ignoring HRESULT or error code can lead to unexpected behavior in error conditions or to low-resource conditions.

LINQ methods are known to not have side effects, and the result should not be ignored.

How to fix violations

If method A creates a new instance of B object that is never used, pass the instance as an argument to another method or assign the instance to a variable. If the object creation is unnecessary, remove the it.

-or-

If method A calls method B, but does not use the new string instance that the method B returns, pass the instance as an argument to another method, assign the instance to a variable. Or remove the call if it is unnecessary.

-or-

If method A calls method B, but does not use the HRESULT or error code that the method returns, use the result in a conditional statement, assign the result to a variable, or pass it as an argument to another method.

-or-

If a LINQ method A calls method B, but does not use the result, use the result in a conditional statement, assign the result to a variable, or pass it as an argument to another method.

When to suppress warnings

Do not suppress a warning from this rule unless the act of creating the object serves some purpose.

Example

The following example shows a class that ignores the result of calling String.Trim.

```
using System;

namespace Samples
{
    public class Book
    {
        private readonly string _Title;

        public Book(string title)
        {
            if (title != null)
            {
                // Violates this rule
                title.Trim();
            }

            _Title = title;
        }

        public string Title
        {
            get { return _Title; }
        }
    }
}
```

```
Imports System

Namespace Samples

    Public Class Book

        Private ReadOnly _Title As String

        Public Sub New(ByVal title As String)

            If title IsNot Nothing Then
                ' Violates this rule
                title.Trim()
            End If

            _Title = title

        End Sub

        Public ReadOnly Property Title() As String
            Get
                Return _Title
            End Get
        End Property

    End Class

End Namespace
```

```
using namespace System;

namespace Samples
{
    public ref class Book
    {
        private:
            initonly String^ _Title;

        public:
        Book(String^ title)
        {
            if (title != nullptr)
            {
                // Violates this rule
                title->Trim();
            }
                _Title = title;
            }

        property String^ Title
        {
            String^ get() { return _Title; }
        }
    };
}
```

Example

The following example fixes the previous violation by assigning the result of String.Trim back to the variable it was called on.

```
using System;

namespace Samples
{
    public class Book
    {
        private readonly string _Title;

        public Book(string title)
        {
            if (title != null)
            {
                title = title.Trim();
            }

            _Title = title;
        }

        public string Title
        {
            get { return _Title; }
        }
    }
}
```

```
Imports System

Namespace Samples

    Public Class Book

        Private ReadOnly _Title As String

        Public Sub New(ByVal title As String)

            If title IsNot Nothing Then
                title = title.Trim()
            End If

            _Title = title

        End Sub

        Public ReadOnly Property Title() As String
            Get
                Return _Title
            End Get
        End Property

    End Class

End Namespace
```

```
using namespace System;

namespace Samples
{
    public ref class Book
    {
    private:
        initonly String^ _Title;

    public:
        Book(String^ title)
        {
            if (title != nullptr)
            {
                title = title->Trim();
            }

            _Title = title;
        }

        property String^ Title
        {
            String^ get() { return _Title; }
        }
    };
}
```

Example

The following example shows a method that does not use an object that it creates.

> **NOTE**
>
> This violation cannot be reproduced in Visual Basic.

```
using namespace System;

namespace Samples
{
    public ref class Book
    {
    public:
        Book()
        {
        }

        static Book^ CreateBook()
        {
            // Violates this rule
            gcnew Book();
            return gcnew Book();
        }
    };
}
```

```
using System;

namespace Samples
{
    public class Book
    {
        public Book()
        {
        }

        public static Book CreateBook()
        {
            // Violates this rule
            new Book();
            return new Book();
        }
    }
}
```

Example

The following example fixes the previous violation by removing the unnecessary creation of an object.

```
using System;

namespace Samples
{
    public class Book
    {
        public Book()
        {
        }

        public static Book CreateBook()
        {
            return new Book();
        }
    }
}
```

```
using namespace System;

namespace Samples
{
    public ref class Book
    {
    public:
        Book()
        {
        }
        static Book^ CreateBook()
        {
            return gcnew Book();
        }
    };
}
```

CA1809: Avoid excessive locals

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1809
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A member contains more than 64 local variables, some of which might be compiler-generated.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A common performance optimization is to store a value in a processor register instead of in memory, which is referred to as *enregistering* the value. The common language runtime considers up to 64 local variables for enregistration. Variables that are not enregistered are put on the stack and must be moved to a register before manipulation. To allow the chance that all local variables get enregistered, limit the number of local variables to 64.

How to fix violations

To fix a violation of this rule, refactor the implementation to use no more than 64 local variables.

When to suppress warnings

It is safe to suppress a warning from this rule, or to disable the rule, if performance is not an issue.

Related rules

CA1804: Remove unused locals

CA1810: Initialize reference type static fields inline

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1810
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A reference type declares an explicit static constructor.

Rule description

When a type declares an explicit static constructor, the just-in-time (JIT) compiler adds a check to each static method and instance constructor of the type to make sure that the static constructor was previously called. Static initialization is triggered when any static member is accessed or when an instance of the type is created. However, static initialization is not triggered if you declare a variable of the type but do not use it, which can be important if the initialization changes global state.

When all static data is initialized inline and an explicit static constructor is not declared, Microsoft intermediate language (MSIL) compilers add the `beforefieldinit` flag and an implicit static constructor, which initializes the static data, to the MSIL type definition. When the JIT compiler encounters the `beforefieldinit` flag, most of the time the static constructor checks are not added. Static initialization is guaranteed to occur at some time before any static fields are accessed but not before a static method or instance constructor is invoked. Note that static initialization can occur at any time after a variable of the type is declared.

Static constructor checks can decrease performance. Often a static constructor is used only to initialize static fields, in which case you must only make sure that static initialization occurs before the first access of a static field. The `beforefieldinit` behavior is appropriate for these and most other types. It is only inappropriate when static initialization affects global state and one of the following is true:

- The effect on global state is expensive and is not required if the type is not used.
- The global state effects can be accessed without accessing any static fields of the type.

How to fix violations

To fix a violation of this rule, initialize all static data when it is declared and remove the static constructor.

When to suppress warnings

It is safe to suppress a warning from this rule if performance is not a concern; or if global state changes that are caused by static initialization are expensive or must be guaranteed to occur before a static method of the type is called or an instance of the type is created.

Example

The following example shows a type, `StaticConstructor`, that violates the rule and a type, `NoStaticConstructor`, that replaces the static constructor with inline initialization to satisfy the rule.

```
using System;
using System.Reflection;
using System.Resources;

namespace PerformanceLibrary
{
   public class StaticConstructor
   {
      static int someInteger;
      static string resourceString;

      static StaticConstructor()
      {
         someInteger = 3;
         ResourceManager stringManager =
            new ResourceManager("strings", Assembly.GetExecutingAssembly());
         resourceString = stringManager.GetString("string");
      }
   }

   public class NoStaticConstructor
   {
      static int someInteger = 3;
      static string resourceString = InitializeResourceString();

      static string InitializeResourceString()
      {
         ResourceManager stringManager =
            new ResourceManager("strings", Assembly.GetExecutingAssembly());
         return stringManager.GetString("string");
      }
   }
}
```

```
Imports System
Imports System.Resources

Namespace PerformanceLibrary

   Public Class StaticConstructor

      Shared someInteger As Integer
      Shared resourceString As String

      Shared Sub New()

         someInteger = 3
         Dim stringManager As New ResourceManager("strings", _
            System.Reflection.Assembly.GetExecutingAssembly())
         resourceString = stringManager.GetString("string")

      End Sub

   End Class


   Public Class NoStaticConstructor

      Shared someInteger As Integer = 3
      Shared resourceString As String = InitializeResourceString()

      Shared Private Function InitializeResourceString()

         Dim stringManager As New ResourceManager("strings", _
            System.Reflection.Assembly.GetExecutingAssembly())
         Return stringManager.GetString("string")

      End Function

   End Class

End Namespace
```

Note the addition of the `beforefieldinit` flag on the MSIL definition for the `NoStaticConstructor` class.

```
.class public auto ansi StaticConstructor
extends [mscorlib]System.Object
{
} // end of class StaticConstructor

.class public auto ansi beforefieldinit NoStaticConstructor
extends [mscorlib]System.Object
{
} // end of class NoStaticConstructor
```

Related rules

- CA2207: Initialize value type static fields inline

CA1811: Avoid uncalled private code

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1811
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A private or internal (assembly-level) member does not have callers in the assembly, is not invoked by the common language runtime, and is not invoked by a delegate. The following members are not checked by this rule:

- Explicit interface members.
- Static constructors.
- Serialization constructors.
- Methods marked with System.Runtime.InteropServices.ComRegisterFunctionAttribute or System.Runtime.InteropServices.ComUnregisterFunctionAttribute.
- Members that are overrides.

Rule description

This rule can report false positives if entry points occur that are not currently identified by the rule logic. Also, a compiler may emit noncallable code into an assembly.

How to fix violations

To fix a violation of this rule, remove the noncallable code or add code that calls it.

When to suppress warnings

It is safe to suppress a warning from this rule. To do so, see Suppress code analysis warnings.

Related rules

CA1812: Avoid uninstantiated internal classes

CA1801: Review unused parameters

CA1804: Remove unused locals

CA1812: Avoid uninstantiated internal classes

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1812
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

An internal (assembly-level) type is never instantiated.

Rule description

This rule tries to locate a call to one of the constructors of the type and reports a violation if no call is found.

The following types are not examined by this rule:

- Value types
- Abstract types
- Enumerations
- Delegates
- Compiler-emitted array types
- Types that cannot be instantiated and that only define `static` (`Shared` in Visual Basic) methods.

If you apply the System.Runtime.CompilerServices.InternalsVisibleToAttribute to the assembly that's being analyzed, this rule will not flag types that are marked as `internal` (`Friend` in Visual Basic) because a field may be used by a friend assembly.

How to fix violations

To fix a violation of this rule, remove the type or add code that uses it. If the type contains only `static` methods, add one of the following to the type to prevent the compiler from emitting a default public instance constructor:

- The `static` modifier for C# types that target .NET Framework 2.0 or later.
- A private constructor for types that target .NET Framework versions 1.0 and 1.1.

When to suppress warnings

It is safe to suppress a warning from this rule. We recommend that you suppress this warning in the following situations:

- The class is created through late-bound reflection methods such as Activator.CreateInstance.
- The class is created automatically by the runtime or ASP.NET. Some examples of automatically created

classes are those that implement System.Configuration.IConfigurationSectionHandler or System.Web.IHttpHandler.

- The class is passed as a type parameter that has a `new` constraint. The following example will be flagged by rule CA1812:

```
internal class MyClass
{
    public void DoSomething()
    {
    }
}
public class MyGeneric<T> where T : new()
{
    public T Create()
    {
        return new T();
    }
}

MyGeneric<MyClass> mc = new MyGeneric<MyClass>();
mc.Create();
```

Related rules

- CA1811: Avoid uncalled private code
- CA1801: Review unused parameters
- CA1804: Remove unused locals

CA1813: Avoid unsealed attributes

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1813
Category	Microsoft.Performance
Breaking change	Breaking

Cause

A public type inherits from System.Attribute, is not abstract, and is not sealed (`NotInheritable` in Visual Basic).

Rule description

.NET provides methods for retrieving custom attributes. By default, these methods search the attribute inheritance hierarchy. For example, System.Attribute.GetCustomAttribute searches for the specified attribute type or any attribute type that extends the specified attribute type. Sealing the attribute eliminates the search through the inheritance hierarchy, and can improve performance.

How to fix violations

To fix a violation of this rule, seal the attribute type or make it abstract.

When to suppress warnings

It is safe to suppress a warning from this rule. Suppress only if you are defining an attribute hierarchy and cannot seal the attribute or make it abstract.

Example

The following example shows a custom attribute that satisfies this rule.

```
using System;

namespace PerformanceLibrary
{
    // Satisfies rule: AvoidUnsealedAttributes.

    [AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct)]
    public sealed class DeveloperAttribute: Attribute
    {
        private string nameValue;
        public DeveloperAttribute(string name)
        {
            nameValue = name;
        }

        public string Name
        {
            get
            {
                return nameValue;
            }
        }
    }

}
```

```
Imports System

Namespace PerformanceLibrary

' Satisfies rule: AvoidUnsealedAttributes.
<AttributeUsage(AttributeTargets.Class Or AttributeTargets.Struct)>  _
NotInheritable Public Class DeveloperAttribute
    Inherits Attribute
    Private nameValue As String

    Public Sub New(name As String)
        nameValue = name
    End Sub


    Public ReadOnly Property Name() As String
        Get
            Return nameValue
        End Get
    End Property
End Class

End Namespace
```

Related rules

- CA1019: Define accessors for attribute arguments
- CA1018: Mark attributes with AttributeUsageAttribute

See also

- Attributes

CA1814: Prefer jagged arrays over multidimensional

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1814
Category	Microsoft.Performance
Breaking change	Breaking

Cause

A member is declared as a multidimensional array.

Rule description

A jagged array is an array whose elements are arrays. The arrays that make up the elements can be of different sizes, leading to less wasted space for some sets of data.

How to fix violations

To fix a violation of this rule, change the multidimensional array to a jagged array.

When to suppress warnings

Suppress a warning from this rule if the multidimensional array does not waste space.

Example

The following example shows declarations for jagged and multidimensional arrays.

```
Imports System

Public Class ArrayHolder
    Private jaggedArray As Integer()() =  {New Integer() {1, 2, 3, 4}, _
                                           New Integer() {5, 6, 7}, _
                                           New Integer() {8}, _
                                           New Integer() {9}}

    Private multiDimArray As Integer(,) =  {{1, 2, 3, 4}, _
                                            {5, 6, 7, 0}, _
                                            {8, 0, 0, 0}, _
                                            {9, 0, 0, 0}}
End Class
```

```
using System;

namespace PerformanceLibrary
{
    public class ArrayHolder
    {
        int[][] jaggedArray = { new int[] {1,2,3,4},
                                new int[] {5,6,7},
                                new int[] {8},
                                new int[] {9}
                              };

        int [,] multiDimArray = {{1,2,3,4},
                                 {5,6,7,0},
                                 {8,0,0,0},
                                 {9,0,0,0}
                                };
    }
}
```

CA1815: Override equals and operator equals on value types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1815
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A value type does not override System.Object.Equals or does not implement the equality operator (==). This rule does not check enumerations.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

For value types, the inherited implementation of Equals uses the Reflection library, and compares the contents of all fields. Reflection is computationally expensive, and comparing every field for equality might be unnecessary. If you expect users to compare or sort instances, or use them as hash table keys, your value type should implement Equals. If your programming language supports operator overloading, you should also provide an implementation of the equality and inequality operators.

How to fix violations

To fix a violation of this rule, provide an implementation of Equals. If you can, implement the equality operator.

When to suppress warnings

It is safe to suppress a warning from this rule if instances of the value type will not be compared to each other.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1815.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Performance). For more information, see Configure FxCop analyzers.

Example

The following code shows a structure (value type) that violates this rule:

```
using System;

namespace Samples
{
    // Violates this rule
    public struct Point
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }
    }
}
```

The following code fixes the previous violation by overriding System.ValueType.Equals and implementing the equality operators (==, !=):

```
using System;

namespace Samples
{
    public struct Point : IEquatable<Point>
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override int GetHashCode()
        {
            return _X ^ _Y;
        }

        public override bool Equals(object obj)
        {
            if (!(obj is Point))
                return false;

            return Equals((Point)obj);
        }

        public bool Equals(Point other)
        {
            if (_X != other._X)
                return false;

            return _Y == other._Y;
        }

        public static bool operator ==(Point point1, Point point2)
        {
            return point1.Equals(point2);
        }

        public static bool operator !=(Point point1, Point point2)
        {
            return !point1.Equals(point2);
        }
    }
}
```

Related rules

- CA2224: Override equals on overloading operator equals
- CA2231: Overload operator equals on overriding ValueType.Equals
- CA2226: Operators should have symmetrical overloads

See also

- System.Object.Equals

CA1816: Call GC.SuppressFinalize correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1816
Category	Microsoft. Usage
Breaking change	Non-breaking

Cause

Violations of this rule can be caused by:

- A method that is an implementation of IDisposable.Dispose and doesn't call GC.SuppressFinalize.
- A method that is not an implementation of IDisposable.Dispose and calls GC.SuppressFinalize.
- A method that calls GC.SuppressFinalize and passes something other than this (C#) or Me (Visual Basic).

Rule description

The IDisposable.Dispose method lets users release resources at any time before the object becoming available for garbage collection. If the IDisposable.Dispose method is called, it frees resources of the object. This makes finalization unnecessary. IDisposable.Dispose should call GC.SuppressFinalize so the garbage collector doesn't call the finalizer of the object.

To prevent derived types with finalizers from having to reimplement IDisposable and to call it, unsealed types without finalizers should still call GC.SuppressFinalize.

How to fix violations

To fix a violation of this rule:

- If the method is an implementation of Dispose, add a call to GC.SuppressFinalize.
- If the method is not an implementation of Dispose, either remove the call to GC.SuppressFinalize or move it to the type's Dispose implementation.
- Change all calls to GC.SuppressFinalize to pass this (C#) or Me (Visual Basic).

When to suppress warnings

Only suppress a warning from this rule if you are deliberately using GC.SuppressFinalize to control the lifetime of other objects. Don't suppress a warning from this rule if an implementation of Dispose doesn't call GC.SuppressFinalize. In this situation, failing to suppress finalization degrades performance and provides no benefits.

Example that violates CA1816

This code shows a method that calls GC.SuppressFinalize, but doesn't pass this (C#) or Me (Visual Basic). As a result,

this code violates rule CA1816.

```
Imports System
Imports System.Data.SqlClient

Namespace Samples

    Public Class DatabaseConnector
        Implements IDisposable

        Private _Connection As New SqlConnection

        Public Sub Dispose() Implements IDisposable.Dispose
            Dispose(True)
            GC.SuppressFinalize(True)   ' Violates rules
        End Sub

        Protected Overridable Sub Dispose(ByVal disposing As Boolean)
            If disposing Then
                If _Connection IsNot Nothing Then
                    _Connection.Dispose()
                    _Connection = Nothing
                End If
            End If
        End Sub

    End Class

End Namespace
```

```
using System;
using System.Data.SqlClient;
namespace Samples
{
    public class DatabaseConnector : IDisposable
    {
        private SqlConnection _Connection = new SqlConnection();

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(true);  // Violates rule
        }

        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                if (_Connection != null)
                {
                    _Connection.Dispose();
                    _Connection = null;
                }
            }
        }
    }
}
```

Example that satisfies CA1816

This example shows a method that correctly calls GC.SuppressFinalize by passing this (C#) or Me (Visual Basic).

```
Imports System
Imports System.Data.SqlClient

Namespace Samples

    Public Class DatabaseConnector
        Implements IDisposable

        Private _Connection As New SqlConnection

        Public Sub Dispose() Implements IDisposable.Dispose
            Dispose(True)
            GC.SuppressFinalize(Me)
        End Sub

        Protected Overridable Sub Dispose(ByVal disposing As Boolean)
            If disposing Then
                If _Connection IsNot Nothing Then
                    _Connection.Dispose()
                    _Connection = Nothing
                End If
            End If
        End Sub

    End Class

End Namespace
```

```
using System;
using System.Data.SqlClient;

namespace Samples
{
    public class DatabaseConnector : IDisposable
    {
        private SqlConnection _Connection = new SqlConnection();

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                if (_Connection != null)
                {
                    _Connection.Dispose();
                    _Connection = null;
                }
            }
        }
    }
}
```

Related rules

- CA2215: Dispose methods should call base class dispose
- CA2216: Disposable types should declare finalizer

See also

- Dispose pattern

CA1819: Properties should not return arrays

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1819
Category	Microsoft.Performance
Breaking change	Breaking

Cause

A property returns an array.

By default, this rule only looks at externally visible properties and types, but this is configurable.

Rule description

Arrays returned by properties are not write-protected, even if the property is read-only. To keep the array tamper-proof, the property must return a copy of the array. Typically, users won't understand the adverse performance implications of calling such a property. Specifically, they might use the property as an indexed property.

How to fix violations

To fix a violation of this rule, either make the property a method or change the property to return a collection.

When to suppress warnings

You can suppress a warning that's raised for a property of an attribute that's derived from the Attribute class. Attributes can contain properties that return arrays, but can't contain properties that return collections.

You can suppress the warning if the property is part of a Data Transfer Object (DTO) class.

Otherwise, do not suppress a warning from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1819.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Performance). For more information, see Configure FxCop analyzers.

Example violation

The following example shows a property that violates this rule:

```
using System;

namespace PerformanceLibrary
{
    public class Book
    {
        private string[] _Pages;

        public Book(string[] pages)
        {
            _Pages = pages;
        }

        public string[] Pages
        {
            get { return _Pages; }
        }
    }
}
```

```
Imports System

Namespace PerformanceLibrary

    Public Class Book

        Private _Pages As String()

        Public Sub New(ByVal pages As String())
            _Pages = pages
        End Sub

        Public ReadOnly Property Pages() As String()
            Get
                Return _Pages
            End Get
        End Property

    End Class

End Namespace
```

To fix a violation of this rule, either make the property a method or change the property to return a collection instead of an array.

Change the property to a method

The following example fixes the violation by changing the property to a method:

```
Imports System

Namespace PerformanceLibrary

    Public Class Book

        Private _Pages As String()

        Public Sub New(ByVal pages As String())
            _Pages = pages
        End Sub

        Public Function GetPages() As String()
            ' Need to return a clone of the array so that consumers
            ' of this library cannot change its contents
            Return DirectCast(_Pages.Clone(), String())
        End Function

    End Class

End Namespace
```

```
using System;

namespace PerformanceLibrary
{
    public class Book
    {
        private string[] _Pages;

        public Book(string[] pages)
        {
            _Pages = pages;
        }

        public string[] GetPages()
        {
            // Need to return a clone of the array so that consumers
            // of this library cannot change its contents
            return (string[])_Pages.Clone();
        }
    }
}
```

Change the property to return a collection

The following example fixes the violation by changing the property to return a System.Collections.ObjectModel.ReadOnlyCollection<T>:

```
using System;
using System.Collections.ObjectModel;

namespace PerformanceLibrary
{
    public class Book
    {
        private ReadOnlyCollection<string> _Pages;
        public Book(string[] pages)
        {
            _Pages = new ReadOnlyCollection<string>(pages);
        }

        public ReadOnlyCollection<string> Pages
        {
            get { return _Pages; }
        }
    }
}
```

```
Imports System
Imports System.Collections.ObjectModel

Namespace PerformanceLibrary

    Public Class Book

        Private _Pages As ReadOnlyCollection(Of String)

        Public Sub New(ByVal pages As String())
            _Pages = New ReadOnlyCollection(Of String)(pages)
        End Sub

        Public ReadOnly Property Pages() As ReadOnlyCollection(Of String)
            Get
                Return _Pages
            End Get
        End Property

    End Class

End Namespace
```

Allow users to modify a property

You might want to allow the consumer of the class to modify a property. The following example shows a read/write property that violates this rule:

```
using System;

namespace PerformanceLibrary
{
    public class Book
    {
        private string[] _Pages;

        public Book(string[] pages)
        {
            _Pages = pages;
        }

        public string[] Pages
        {
            get { return _Pages; }
            set { _Pages = value; }
        }
    }
}
```

```
Imports System

Namespace PerformanceLibrary

    Public Class Book

        Private _Pages As String()

        Public Sub New(ByVal pages As String())
            _Pages = pages
        End Sub

        Public Property Pages() As String()
            Get
                Return _Pages
            End Get

            Set(ByVal value as String())
                _Pages = value
            End Set
        End Property

    End Class

End Namespace
```

The following example fixes the violation by changing the property to return a System.Collections.ObjectModel.Collection<T>:

```
Imports System
Imports System.Collections.ObjectModel

Namespace PerformanceLibrary

    Public Class Book

        Private _Pages As Collection(Of String)

        Public Sub New(ByVal pages As String())
            _Pages = New Collection(Of String)(pages)
        End Sub

        Public ReadOnly Property Pages() As Collection(Of String)
            Get
                Return _Pages
            End Get
        End Property

    End Class

End Namespace
```

```
using System;
using System.Collections.ObjectModel;

namespace PerformanceLibrary
{
    public class Book
    {
        private Collection<string> _Pages;

        public Book(string[] pages)
        {
            _Pages = new Collection<string>(pages);
        }

        public Collection<string> Pages
        {
            get { return _Pages; }
        }
    }
}
```

Related rules

- CA1024: Use properties where appropriate

CA1820: Test for empty strings using string length

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1820
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A string is compared to the empty string by using Object.Equals.

Rule description

Comparing strings using the String.Length property or the String.IsNullOrEmpty method is faster than using Equals. This is because Equals executes significantly more MSIL instructions than either IsNullOrEmpty or the number of instructions executed to retrieve the Length property value and compare it to zero.

For null strings, Equals and `<string>.Length == 0` behave differently. If you try to get the value of the Length property on a null string, the common language runtime throws a System.NullReferenceException. If you perform a comparison between a null string and the empty string, the common language runtime does not throw an exception and returns `false`. Testing for null does not significantly affect the relative performance of these two approaches. When targeting .NET Framework 2.0 or later, use the IsNullOrEmpty method. Otherwise, use the Length == 0 comparison whenever possible.

How to fix violations

To fix a violation of this rule, change the comparison to use the IsNullOrEmpty method.

When to suppress warnings

It's safe to suppress a warning from this rule if performance is not an issue.

Example

The following example illustrates the different techniques that are used to look for an empty string.

```
using System;

namespace PerformanceLibrary
{
   public class StringTester
   {
      string s1 = "test";

      public void EqualsTest()
      {
         // Violates rule: TestForEmptyStringsUsingStringLength.
         if (s1 == "")
         {
            Console.WriteLine("s1 equals empty string.");
         }
      }

      // Use for .NET Framework 1.0 and 1.1.
      public void LengthTest()
      {
         // Satisfies rule: TestForEmptyStringsUsingStringLength.
         if (s1 != null && s1.Length == 0)
         {
            Console.WriteLine("s1.Length == 0.");
         }
      }

      // Use for .NET Framework 2.0.
      public void NullOrEmptyTest()
      {
         // Satisfies rule: TestForEmptyStringsUsingStringLength.
         if ( !String.IsNullOrEmpty(s1) )
         {
            Console.WriteLine("s1 != null and s1.Length != 0.");
         }
      }
   }
}
```

CA1821: Remove empty finalizers

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1821
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

A type implements a finalizer that is empty, calls only the base type finalizer, or calls only conditionally emitted methods.

Rule description

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.

How to fix violations

Remove the empty finalizer. If a finalizer is required for debugging, enclose the whole finalizer in `#if DEBUG / #endif` directives.

When to suppress warnings

Do not suppress a message from this rule.

Example

The following example shows an empty finalizer that should be removed, a finalizer that should be enclosed in `#if DEBUG / #endif` directives, and a finalizer that uses the `#if DEBUG / #endif` directives correctly.

```
using System.Diagnostics;

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!");
    }
}

public class Class3
{
    #if DEBUG
        // Violation will not occur because the finalizer will exist and
        // contain code when the DEBUG directive is present. When the
        // DEBUG directive is not present, the finalizer will not exist,
        // and therefore not be empty.
        ~Class3()
        {
            Debug.Fail("Finalizer called!");
        }
    #endif
}
```

CA1822: Mark members as static

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1822
Category	Microsoft.Performance
Breaking change	Non-breaking - If the member is not visible outside the assembly, regardless of the change you make. Non-breaking - If you just change the member to an instance member with the `this` keyword. Breaking - If you change the member from an instance member to a static member and it is visible outside the assembly.

Cause

A member that does not access instance data is not marked as static (Shared in Visual Basic).

Rule description

Members that do not access instance data or call instance methods can be marked as static (Shared in Visual Basic). After you mark the methods as static, the compiler will emit nonvirtual call sites to these members. Emitting nonvirtual call sites will prevent a check at run time for each call that makes sure that the current object pointer is non-null. This can achieve a measurable performance gain for performance-sensitive code. In some cases, the failure to access the current object instance represents a correctness issue.

How to fix violations

Mark the member as static (or Shared in Visual Basic) or use 'this'/'Me' in the method body, if appropriate.

When to suppress warnings

It is safe to suppress a warning from this rule for previously shipped code for which the fix would be a breaking change.

Related rules

CA1811: Avoid uncalled private code

CA1812: Avoid uninstantiated internal classes

CA1804: Remove unused locals

CA1823: Avoid unused private fields

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1823
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

This rule is reported when a private field in your code exists but is not used by any code path.

Rule description

Private fields were detected that do not appear to be accessed in the assembly.

How to fix violations

To fix a violation of this rule, remove the field or add code that uses it.

When to suppress warnings

It is safe to suppress a warning from this rule.

Related rules

CA1812: Avoid uninstantiated internal classes

CA1801: Review unused parameters

CA1804: Remove unused locals

CA1811: Avoid uncalled private code

CA1824: Mark assemblies with NeutralResourcesLanguageAttribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1824
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

An assembly contains a **ResX**-based resource but does not have the System.Resources.NeutralResourcesLanguageAttribute applied to it.

Rule description

The NeutralResourcesLanguageAttribute attribute informs the resource manager of an app's default culture. If the default culture's resources are embedded in the app's main assembly, and ResourceManager has to retrieve resources that belong to the same culture as the default culture, the ResourceManager automatically uses the resources located in the main assembly instead of searching for a satellite assembly. This bypasses the usual assembly probe, improves lookup performance for the first resource you load, and can reduce your working set.

> **TIP**
>
> See Packaging and deploying resources for the process that ResourceManager uses to probe for resource files.

Fix violations

To fix a violation of this rule, add the attribute to the assembly, and specify the language of the resources of the neutral culture.

To specify the neutral language for resources

1. In **Solution Explorer**, right-click your project, and then select **Properties**.
2. Select the **Application** tab, and then select **Assembly Information**.

   > **NOTE**
   >
   > If your project is a .NET Standard or .NET Core project, select the **Package** tab.

3. Select the language from the **Neutral language** or **Assembly neutral language** drop-down list.
4. Select **OK**.

When to suppress warnings

It is permissible to suppress a warning from this rule. However, startup performance might degrade.

See also

- NeutralResourcesLanguageAttribute
- Resources in desktop apps (.NET)
- CA1703 - Resource strings should be spelled correctly
- CA1701 - Resource string compound words should be cased correctly

CA1825: Avoid zero-length array allocations

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1825
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

An empty Array with no elements is allocated.

Rule description

Initializing a zero-length array leads to an unnecessary memory allocation. Instead, use the statically allocated empty array instance by calling the Array.Empty method. The memory allocation is shared across all invocations of this method.

> **NOTE**
> Rule CA1825 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.6.0.

How to fix violations

To fix a violation, replace the zero-length array allocation with a call to Array.Empty. For example, the following two code snippets show a violation of the rule and how to fix it:

```
class C
{
    public void M1()
    {
        // Violates rule CA1825.
        var a = new int[0];
    }
}
```

```
class C
{
    public void M1()
    {
        // Resolves rule CA1825 violation.
        var a = System.Array.Empty<int>();
    }
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the array allocation and press **Ctrl+.** (period). Choose **Use Array.Empty** from the list of options that's presented.

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about the additional memory allocation.

Related rules

- CA1814: Prefer jagged arrays over multidimensional

See also

- Performance warnings

CA1826: Use property instead of Linq Enumerable method

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1826
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

The Enumerable LINQ method was used on a type that supports an equivalent, more efficient property.

Rule description

This rule flags the Enumerable LINQ method calls on collections of types that have equivalent, but more efficient properties to fetch the same data.

This rule analyzes the following collection types:

- A type that implements IReadOnlyList<T>, but not IList<T>

This rule flags calls to following methods on these collection types:

- System.Linq.Enumerable.Count method
- System.Linq.Enumerable.First method
- System.Linq.Enumerable.FirstOrDefault method
- System.Linq.Enumerable.Last method
- System.Linq.Enumerable.LastOrDefault method

The analyzed collection types and/or methods may be extended in future to cover more cases.

> **NOTE**
> Rule CA1826 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.6.

How to fix violations

To fix a violation, replace the Enumerable method calls with property access. For example, the following two code snippets show a violation of the rule and how to fix it:

```
using System;
using System.Collections.Generic;
using System.Linq;

class C
{
    public void M(IReadOnlyList<string> list)
    {
        Console.Write(list.First());
        Console.Write(list.Last());
        Console.Write(list.Count());
    }
}
```

```
using System;
using System.Collections.Generic;

class C
{
    public void M(IReadOnlyList<string> list)
    {
        Console.Write(list[0]);
        Console.Write(list[list.Count - 1]);
        Console.Write(list.Count);
    }
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the violation and press **Ctrl+.** (period). Choose **Use indexer** from the list of options that's presented.

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about the performance impact from specific Enumerable method calls.

Related rules

- CA1827: Do not use Count/LongCount when Any can be used
- CA1828: Do not use CountAsync/LongCountAsync when AnyAsync can be used
- CA1829: Use Length/Count property instead of Enumerable.Count method

See also

- Performance warnings

CA1827: Do not use Count/LongCount when Any can be used

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1827
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

The Count or LongCount method was used where Any method would be more efficient.

Rule description

This rule flags the Count and LongCount LINQ method calls used to check if the collection has at least one element. These method calls require enumerating the entire collection to compute the count. The same check is faster with the Any method as it avoids enumerating the collection.

> **NOTE**
>
> Rule CA1827 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.6.

How to fix violations

To fix a violation, replace the Count or LongCount method call with the Any method. For example, the following two code snippets show a violation of the rule and how to fix it:

```
using System.Collections.Generic;
using System.Linq;

class C
{
    public string M1(IEnumerable<string> list)
        => list.Count() != 0 ? "Not empty" : "Empty";

    public string M2(IEnumerable<string> list)
        => list.LongCount() > 0 ? "Not empty" : "Empty";
}
```

```
using System.Collections.Generic;
using System.Linq;

class C
{
    public string M1(IEnumerable<string> list)
        => list.Any() ? "Not empty" : "Empty";

    public string M2(IEnumerable<string> list)
        => list.Any() ? "Not empty" : "Empty";
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the array violation and press **Ctrl+.** (period). Choose **Do not use Count() or LongCount() when Any() can be used** from the list of options that's presented.

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about the performance impact from unnecessary collection enumeration to compute the count.

Related rules

- CA1826: Use property instead of Linq Enumerable method
- CA1828: Do not use CountAsync/LongCountAsync when AnyAsync can be used
- CA1829: Use Length/Count property instead of Enumerable.Count method

See also

- Performance warnings

CA1828: Do not use CountAsync/LongCountAsync when AnyAsync can be used

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1828
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

CountAsync or The LongCountAsync method was used where the AnyAsync method would be more efficient.

Rule description

This rule flags the CountAsync and LongCountAsync LINQ method calls used to check if the collection has at least one element. These method calls require enumerating the entire collection to compute the count. The same check is faster with the AnyAsync method as it avoids enumerating the collection.

> **NOTE**
>
> Rule CA1828 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.6.

How to fix violations

To fix a violation, replace the CountAsync or LongCountAsync method call with the AnyAsync method. For example, the following two code snippets show a violation of the rule and how to fix it:

```
using System.Linq;
using System.Threading.Tasks;
using static Microsoft.EntityFrameworkCore.EntityFrameworkQueryableExtensions;

class C
{
    public async Task<string> M1Async(IQueryable<string> list)
        => await list.CountAsync() != 0 ? "Not empty" : "Empty";

    public async Task<string> M2Async(IQueryable<string> list)
        => await list.LongCountAsync() > 0 ? "Not empty" : "Empty";
}
```

```
using System.Linq;
using System.Threading.Tasks;
using static Microsoft.EntityFrameworkCore.EntityFrameworkQueryableExtensions;

class C
{
    public async Task<string> M1Async(IQueryable<string> list)
        => await list.AnyAsync() ? "Not empty" : "Empty";

    public async Task<string> M2Async(IQueryable<string> list)
        => await list.AnyAsync() ? "Not empty" : "Empty";
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the violation and press **Ctrl+.** (period). Choose **Do not use CountAsync() or LongCountAsync() when AnyAsync() can be used** from the list of options that's presented.

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about the performance impact from unnecessary collection enumeration to compute the count.

Related rules

- CA1826: Use property instead of Linq Enumerable method
- CA1827: Do not use Count/LongCount when Any can be used
- CA1829: Use Length/Count property instead of Enumerable.Count method

See also

- Performance warnings

CA1829: Use Length/Count property instead of Enumerable.Count method

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1829
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

The Count LINQ method was used on a type that supports an equivalent, more efficient `Length` or `Count` property.

Rule description

This rule flags the Count LINQ method calls on collections of types that have equivalent, but more efficient `Length` or `Count` property to fetch the same data. `Length` or `Count` property does not enumerate the collection, hence is more efficient.

This rule flags Count calls on the following collection types with `Length` property:

- System.Array
- System.Collections.Immutable.ImmutableArray<T>

This rule flags Count calls on the following collection types with the `Count` property:

- System.Collections.ICollection
- System.Collections.Generic.ICollection<T>
- System.Collections.Generic.IReadOnlyCollection<T>

The analyzed collection types may be extended in future to cover more cases.

> **NOTE**
> Rule CA1829 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.6.

How to fix violations

To fix a violation, replace the Count method call with use of the `Length` or `Count` property access. For example, the following two code snippets show a violation of the rule and how to fix it:

```
using System.Collections.Generic;
using System.Linq;

class C
{
    public int GetCount(int[] array)
        => array.Count();

    public int GetCount(ICollection<int> collection)
        => collection.Count();
}
```

```
using System.Collections.Generic;

class C
{
    public int GetCount(int[] array)
        => array.Length;

    public int GetCount(ICollection<int> collection)
        => collection.Count;
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the violation and press **Ctrl+.** (period). Choose **Use Length/Count property instead of Count() when available** from the list of options that's presented.

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about the performance impact from unnecessary collection enumeration to compute the count.

Related rules

- CA1826: Use property instead of Linq Enumerable method
- CA1827: Do not use Count/LongCount when Any can be used
- CA1828: Do not use CountAsync/LongCountAsync when AnyAsync can be used

See also

- Performance warnings

CA1830: Prefer strongly-typed Append and Insert method overloads on StringBuilder.

8/4/2020 • 2 minutes to read • Edit Online

CheckId	CA1830
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

An StringBuilder `Append` or `Insert` method was called with an argument that was the result of calling `ToString` on a type for which the `Append` or `Insert` method has a dedicated overload.

Rule description

Append and Insert provide overloads for multiple types beyond String. When possible, prefer the strongly-typed overloads over using ToString() and the string-based overload.

How to fix violations

Delete the unnecessary `ToString()` from the invocation.

```
using System.Text;

class C
{
    int _value;

    // Violation
    public void Log(StringBuilder destination)
    {
        destination.Append("Value: ").Append(_value.ToString()).AppendLine();
    }

    // Fixed
    public void Log(StringBuilder destination)
    {
        destination.Append("Value: ").Append(_value).AppendLine();
    }
}
```

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about the performance impact from unnecessary string allocations.

See also

- Performance warnings

CA1831: Use AsSpan instead of Range-based indexers for string when appropriate

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1831
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

When using a range-indexer on a string and implicitly assigning the value to ReadOnlySpan<char>.

Rule description

Using a range-indexer on a string and assigning to a span type: The range indexer on a Span<T> is a non-copying Slice operation, but for the range indexer on a string the method Substring will be used instead of Slice, which produces a copy of requested portion of the string. This copy is usually unnecessary when it is implicitly used as a ReadOnlySpan<T> or ReadOnlyMemory<T> value. If a copy isn't intended, use the AsSpan method to avoid the unnecessary copy. If the copy is intended, either assign it to a local variable first or add an explicit cast. The analyzer only reports when an implicit cast is used on the result of the range indexer operation.

Detects

Implicit conversions:

- `ReadOnlySpan<char> slice = str[a..b];`

Does not detect

Explicit conversions:

- `ReadOnlySpan<char> slice = (ReadOnlySpan<char>)str[a..b];`

How to fix violations

To fix the violation of this rule: use AsSpan instead of the Range-based indexer on the string to avoid creating unnecessary data copies.

```
class C
{
    public void TestMethod(string str)
    {
        // The violation occurs
        ReadOnlySpan<char> slice = str[1..3];
        ...
    }
}
```

```
class C
{
    public void TestMethod(string str)
    {
        // The violation fixed with AsSpan extension method
        ReadOnlySpan<char> slice = str.AsSpan()[1..3];
        ...
    }
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the array violation and press **Ctrl+.** (period). Choose **Use AsSpan instead of the Range-based indexer on a string.** from the list of options that is presented.

When to suppress warnings

It's safe to suppress a violation of this rule if creating a copy is intended. To suppress the warning, you can simply add an explicit cast.

```
class C
{
    public void TestMethod(string str)
    {
        // The violation occurs
        ReadOnlySpan<char> slice = str[1..3];
        ...
    }
}
```

```
class C
{
    public void TestMethod(string str)
    {
        // The violation fixed with explicit casting
        ReadOnlySpan<char> slice = (ReadOnlySpan<char>)str[1..3];
        ...
    }
}
```

Related rules

- CA1832: Use AsSpan or AsMemory instead of Range-based indexers for getting ReadOnlySpan or

ReadOnlyMemory portion of an array

- CA1833: Use AsSpan or AsMemory instead of Range-based indexers for getting Span or Memory portion of an array

See also

- Performance warnings

CA1832: Use AsSpan or AsMemory instead of Range-based indexers for getting ReadOnlySpan or ReadOnlyMemory portion of an array

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1832
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

When using a range-indexer on an array and implicitly assigning the value to ReadOnlySpan<T> or ReadOnlyMemory<T>.

Rule description

Using a range-indexer on an array and assigning to a memory or span type: The range indexer on a Span<T> is a non-copying Slice operation, but for the range indexer on array the method GetSubArray will be used instead of Slice, which produces a copy of requested portion of the array. This copy is usually unnecessary when it is implicitly used as a ReadOnlySpan<T> or ReadOnlyMemory<T> value. If a copy isn't intended, use the AsSpan or AsMemory method to avoid the unnecessary copy. If the copy is intended, either assign it to a local variable first or add an explicit cast. The analyzer only reports when an implicit cast is used on the result of the range indexer operation.

Detects

Implicit conversions:

- `ReadOnlySpan<SomeT> slice = arr[a..b];`
- `ReadOnlyMemory<SomeT> slice = arr[a..b];`

Does not detect

Explicit conversions:

- `ReadOnlySpan<SomeT> slice = (ReadOnlySpan<SomeT>)arr[a..b];`
- `ReadOnlyMemory<SomeT> slice = (ReadOnlyMemory<SomeT>)arr[a..b];`

How to fix violations

To fix the violation of this rule: use the AsSpan or AsMemory extension method to avoid creating unnecessary data copies.

```
class C
{
    public void TestMethod(byte[] arr)
    {
        // The violation occurs for both statements below
        ReadOnlySpan<byte> tmp1 = arr[0..2];
        ReadOnlyMemory<byte> tmp3 = arr[5..8];
        ...
    }
}
```

```
class C
{
    public void TestMethod(byte[] arr)
    {
        // The violations fixed with AsSpan or AsMemory accordingly
        ReadOnlySpan<byte> tmp1 = arr.AsSpan()[0..2];
        ReadOnlyMemory<byte> tmp3 = arr.AsMemory()[5..8];
        ...
    }
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the array violation and press **Ctrl+.** (period). Choose **Use AsSpan instead of the Range-based indexer on an array** from the list of options that is presented.

When to suppress warnings

It's safe to suppress a violation of this rule if creating a copy is intended. To suppress this warning, you can simply add an explicit cast.

```
class C
{
    public void TestMethod(byte arr[])
    {
        // The violation occurs
        ReadOnlySpan<byte> tmp1 = arr[0..2];
        ReadOnlyMemory<byte> tmp3 = arr[5..8];
        ...
    }
}
```

```
class C
{
    public void TestMethod(byte arr[])
    {
        // The violation fixed with explicit casting
        ReadOnlySpan<byte> tmp1 = (ReadOnlySpan<byte>)arr[0..2];
        ReadOnlyMemory<byte> tmp3 = (ReadOnlyMemory<byte>)arr[5..8];
        ...
    }
}
```

Related rules

- CA1831: Use AsSpan instead of Range-based indexers for string when appropriate
- CA1833: Use AsSpan or AsMemory instead of Range-based indexers for getting Span or Memory portion of an array

See also

- Performance warnings

CA1833: Use AsSpan or AsMemory instead of Range-based indexers for getting Span or Memory portion of an array

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1833
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

When using a range-indexer on an array and implicitly assigning the value to Span<T> or Memory<T>.

Rule description

Using a range-indexer on array and assigning to a memory or span type: The range indexer on a Span<T> is a non-copying Slice operation, but for the range indexer on array the method GetSubArray will be used instead of Slice, which produces a copy of requested portion of the array. This copy is usually unnecessary when it is implicitly used as a Span<T> or Memory<T> value. If a copy isn't intended, use the AsSpan or AsMemory method to avoid the unnecessary copy. If the copy is intended, either assign it to a local variable first or add an explicit cast. The analyzer only reports when an implicit cast is used on the result of the range indexer operation.

Detects

Implicit conversions:

- `Span<SomeT> slice = arr[a..b];`
- `Memory<SomeT> slice = arr[a..b];`

Does not detect

Explicit conversions:

- `Span<SomeT> slice = (Span<SomeT>)arr[a..b];`
- `Memory<SomeT> slice = (Memory<SomeT>)arr[a..b];`

How to fix violations

To fix the violation of this rule: use the AsSpan or AsMemory extension method to avoid creating unnecessary data copies.

```
class C
{
    public void TestMethod(byte[] arr)
    {
        // The violation occurs for both statements below
        Span<byte> tmp2 = arr[0..5];
        Memory<byte> tmp4 = arr[5..10];
        ...
    }
}
```

```
class C
{
    public void TestMethod(byte[] arr)
    {
        // The violations fixed with AsSpan or AsMemory accordingly
        Span<byte> tmp2 = arr.AsSpan()[0..5];
        Memory<byte> tmp4 = arr.AsMemory()[5..10];
        ...
    }
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the array violation and press **Ctrl+.** (period). Choose **Use AsMemory instead of the Range-based indexer on an array** from the list of options that is presented.

When to suppress warnings

It's safe to suppress a violation of this rule if creating a copy is intended. To suppress this warning, you can simply add an explicit cast.

```
class C
{
    public void TestMethod(byte arr[])
    {
        // The violation occurs
        Span<byte> tmp1 = arr[0..5];
        Memory<byte> tmp2 = arr[5..10];
  ...
    }
}
```

```
class C
{
    public void TestMethod(byte arr[])
    {
        // The violation fixed with explicit casting
        Span<byte> tmp1 = (Span<byte>)arr[0..5];
        Memory<byte> tmp2 = (Memory<byte>)arr[5..10];
        ...
    }
}
```

Related rules

- CA1831: Use AsSpan instead of Range-based indexers for string when appropriate
- CA1832: Use AsSpan or AsMemory instead of Range-based indexers for getting ReadOnlySpan or ReadOnlyMemory portion of an array

See also

- Performance warnings

CA1835: Prefer the memory-based overloads of ReadAsync/WriteAsync methods in stream-based classes

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
TypeName	PreferStreamAsyncMemoryOverlodas
CheckId	CA1835
Category	Microsoft.Performance
Breaking Change	Non-breaking

Cause

This rule locates awaited invocations of the byte-array-based method overloads for `ReadAsync` and `WriteAsync`, and suggests using the memory-based method overloads instead, because they are more efficient.

Rule description

The memory-based method overloads have a more efficient memory usage than the byte array-based ones.

The rule works on `ReadAsync` and `WriteAsync` invocations of any class that inherits from Stream.

The rule only works when the method is preceded by the `await` keyword.

DETECTED METHOD	SUGGESTED METHOD
ReadAsync(Byte[], Int32, Int32, CancellationToken)	ReadAsync(Memory<Byte>, CancellationToken)
ReadAsync(Byte[], Int32, Int32)	ReadAsync(Memory<Byte>, CancellationToken) with `CancellationToken` set to `default` in C#, or `Nothing` in Visual Basic.
WriteAsync(Byte[], Int32, Int32, CancellationToken)	WriteAsync(ReadOnlyMemory<Byte>, CancellationToken)
WriteAsync(Byte[], Int32, Int32, CancellationToken)	WriteAsync(ReadOnlyMemory<Byte>, CancellationToken) with `CancellationToken` set to `default` in C#, or `Nothing` in Visual Basic.

IMPORTANT

Make sure to pass the `offset` and `count` integer arguments to the created `Memory` or `ReadOnlyMemory` instances.

> **NOTE**
>
> Rule CA1835 is available in all .NET versions where the memory-based overloads are available:
>
> - .NET Standard 2.1 and above.
> - .NET Core 2.1 and above.

How to fix violations

You can either fix them manually, or you can opt to let Visual Studio do it for you, by hovering over the light bulb that shows up next to the method invocation, and selecting the suggested change. Example:

The rule can detect a variety of violations for the `ReadAsync` and `WriteAsync` methods. Here are examples of the cases that the rule can detect:

Example 1

Invocations of `ReadAsync`, without and with a `CancellationToken` argument:

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod(CancellationToken ct)
    {
        using (FileStream s = new FileStream("path.txt", FileMode.Create))
        {
            byte[] buffer = new byte[s.Length];
            await s.ReadAsync(buffer, 0, buffer.Length);
            await s.ReadAsync(buffer, 0, buffer.Length, ct);
        }
    }
}
```

Fix

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod(CancellationToken ct)
    {
        using (FileStream s = new FileStream("path.txt", FileMode.Create))
        {
            byte[] buffer = new byte[s.Length];
            await s.ReadAsync(buffer.AsMemory(0, buffer.Length));
            await s.ReadAsync(buffer.AsMemory(0, buffer.Length), ct);
        }
    }
}
```

Example 2

Invocations of `WriteAsync`, without and with a `CancellationToken` argument:

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod(CancellationToken ct)
    {
        using (FileStream s = File.Open("path.txt", FileMode.Open))
        {
            byte[] buffer = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
            await s.WriteAsync(buffer, 0, buffer.Length);
            await s.WriteAsync(buffer, 0, buffer.Length, ct);
        }
    }
}
```

Fix

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod()
    {
        using (FileStream s = File.Open("path.txt", FileMode.Open))
        {
            byte[] buffer = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
            await s.WriteAsync(buffer.AsMemory(0, buffer.Length));
            await s.WriteAsync(buffer.AsMemory(0, buffer.Length), ct);
        }
    }
}
```

Example 3

Invocations with `ConfigureAwait`:

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod()
    {
        using (FileStream s = File.Open("path.txt", FileMode.Open))
        {
            byte[] buffer1 = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
            await s.WriteAsync(buffer1, 0, buffer1.Length).ConfigureAwait(false);

            byte[] buffer2 = new byte[s.Length];
            await s.ReadAsync(buffer2, 0, buffer2.Length).ConfigureAwait(true);
        }
    }
}
```

Fix

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod()
    {
        using (FileStream s = File.Open("path.txt", FileMode.Open))
        {
            byte[] buffer1 = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
            await s.WriteAsync(buffer1.AsMemory(0, buffer1.Length)).ConfigureAwait(false);

            byte[] buffer2 = new byte[s.Length];
            await s.ReadAsync(buffer2.AsMemory(0, buffer.Length)).ConfigureAwait(true);
        }
    }
}
```

No diagnosis

Here are some examples of invocations where the rule will **not** be fired:

Example 1

The return value is saved in a `Task` variable instead of being awaited:

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

class MyClass
{
    public void MyMethod()
    {
        byte[] buffer = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
        using (FileStream s = new FileStream("path.txt", FileMode.Create))
        {
            Task t = s.WriteAsync(buffer, 0, buffer.Length);
        }
    }
}
```

Example 2

The return value is returned by the wrapping method instead of being awaited:

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

class MyClass
{
    public Task MyMethod(FileStream s, byte[] buffer)
    {
        return s.WriteAsync(buffer, 0, buffer.Length);
    }
}
```

Example 3

The return value is used to call `ContinueWith`, which is the method being awaited:

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

class MyClass
{
    public void MyMethod()
    {
        byte[] buffer = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
        using (FileStream s = new FileStream("path.txt", FileMode.Create))
        {
            await s.WriteAsync(buffer, 0, buffer.Length).ContinueWith(c => { /* ... */ });
        }
    }
}
```

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about improving performance when reading or writing buffers in stream-based classes.

See also

- Performance warnings

CA1836: Prefer IsEmpty over Count when available

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1836
Category	Microsoft.Performance
Breaking change	Non-breaking

Cause

The `Count` or `Length` property or the Count<TSource>(IEnumerable<TSource>) extension method was used to determine whether or not the object contains any items by comparing the value to `0` or `1`, and the object has a more efficient `IsEmpty` property that could be used instead.

Rule description

This rule flags the calls to the `Count` and `Length` properties or Count<TSource>(IEnumerable<TSource>) and LongCount<TSource>(IEnumerable<TSource>) LINQ methods when they are used to determine if the object contains any items and the object has a more efficient `IsEmpty` property.

The analysis of this rule originally overlapped with similar rules CA1827, CA1828, and CA1829; the analyzers of such rules were merged along with the one for CA1836 to report the best diagnosis in case of overlap.

How to fix violations

To fix a violation, replace the Count<TSource>(IEnumerable<TSource>) or LongCount<TSource>(IEnumerable<TSource>) method call or the `Length` or `Count` property access when it's used in an operation that determines if the object is empty with use of the `IsEmpty` property access. For example, the following two code snippets show a violation of the rule and how to fix it:

```
using System.Collections.Concurrent;

class C
{
    ConcurrentQueue<int> _queue;
    public bool IsEmpty => _queue.Count == 0;
}
```

```
using System.Collections.Concurrent;

class C
{
    ConcurrentQueue<int> _queue;
    public bool IsEmpty => _queue.IsEmpty;
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the violation and press **Ctrl+.** (period). Choose **Prefer 'IsEmpty' over 'Count' to determine whether the object contains or not any items** from the list of options that's presented.

When to suppress warnings

It's safe to suppress a violation of this rule if you're not concerned about the performance impact from unnecessary item enumeration to compute the count.

Related rules

- CA1827: Do not use Count/LongCount when Any can be used
- CA1828: Do not use CountAsync/LongCountAsync when AnyAsync can be used
- CA1829: Do not use CountAsync/LongCountAsync when AnyAsync can be used

See also

- Performance warnings

Portability Warnings

2/28/2020 • 2 minutes to read • Edit Online

Portability warnings support portability across different operating systems.

In This Section

RULE	DESCRIPTION
CA1900: Value type fields should be portable	This rule checks that structures that are declared by using an explicit layout attribute will align correctly when marshaled to unmanaged code on 64-bit operating systems.
CA1901: P/Invoke declarations should be portable	This rule evaluates the size of each parameter and the return value of a P/Invoke, and verifies that their size is correct when marshaled to unmanaged code on 32-bit and 64-bit operating systems.
CA1903: Use only API from targeted framework	A member or type is using a member or type that was introduced in a service pack that was not included together with the targeted framework of the project.

CA1900: Value type fields should be portable

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1900
Category	Microsoft.Portability
Breaking change	Breaking - If the field can be seen outside the assembly. Non-breaking - If the field is not visible outside the assembly.

Cause

This rule checks that structures that are declared with explicit layout will align correctly when marshaled to unmanaged code on 64-bit operating systems. IA-64 does not allow unaligned memory accesses and the process will crash if this violation is not fixed.

Rule description

Structures that have explicit layout that contains misaligned fields cause crashes on 64-bit operating systems.

How to fix violations

All fields that are smaller than 8 bytes must have offsets that are a multiple of their size, and fields that are 8 bytes or more must have offsets that are a multiple of 8. Another solution is to use `LayoutKind.Sequential` instead of `LayoutKind.Explicit`, if reasonable.

When to suppress warnings

This warning should be suppressed only if it occurs in error.

CA1901: P/Invoke declarations should be portable

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1901
Category	Microsoft.Portability
Breaking change	Breaking - If the P/Invoke is visible outside the assembly. Non-breaking - If the P/Invoke is not visible outside the assembly.

Cause

This rule evaluates the size of each parameter and the return value of a P/Invoke and verifies that their size, when marshaled to unmanaged code on 32-bit and 64-bit platforms, is correct. The most common violation of this rule is to pass a fixed-sized integer where a platform-dependent, pointer-sized variable is required.

> **NOTE**
>
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Either of the following scenarios violates this rule occurs:

- The return value or parameter is typed as a fixed-size integer when it should be typed as an `IntPtr`.
- The return value or parameter is typed as an `IntPtr` when it should be typed as a fixed-size integer.

How to fix violations

You can fix this violation by using `IntPtr` or `UIntPtr` to represent handles instead of `Int32` or `UInt32`.

When to suppress warnings

You should not suppress this warning.

Example

The following example demonstrates a violation of this rule.

```
internal class NativeMethods
{
    [DllImport("shell32.dll", CharSet=CharSet.Auto)]
    internal static extern IntPtr ExtractIcon(IntPtr hInst,
        string lpszExeFileName, IntPtr nIconIndex);
}
```

In this example, the `nIconIndex` parameter is declared as an `IntPtr`, which is 4 bytes wide on a 32-bit platform and 8 bytes wide on a 64-bit platform. In the unmanaged declaration that follows, you can see that `nIconIndex` is a 4-byte unsigned integer on all platforms.

```
HICON ExtractIcon(HINSTANCE hInst, LPCTSTR lpszExeFileName,
    UINT nIconIndex);
```

Example

To fix the violation, change the declaration to the following:

```
internal class NativeMethods{
    [DllImport("shell32.dll", CharSet=CharSet.Auto)]
    internal static extern IntPtr ExtractIcon(IntPtr hInst,
        string lpszExeFileName, uint nIconIndex);
}
```

See also

Portability Warnings

CA1903: Use only API from targeted framework

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1903
Category	Microsoft.Portability
Breaking change	Breaking - when fired against the signature of an externally visible member or type. Non-Breaking - when fired in the body of a method.

Cause

A member or type is using a member or type that was introduced in a service pack that was not included with the project's targeted framework.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

New members and types were included in .NET Framework 2.0 Service Pack 1 and 2, .NET Framework 3.0 Service Pack 1 and 2, and .NET Framework 3.5 Service Pack 1. Projects that target the major versions of the .NET Framework can unintentionally take dependencies on these new APIs. To prevent this dependency, this rule fires on usages of any new members and types that were not included by default with the project's target framework.

Target Framework and Service Pack Dependencies

ITEM	VALUE
When target framework is	Fires on usages of members introduced in
.NET Framework 2.0	.NET Framework 2.0 SP1, .NET Framework 2.0 SP2
.NET Framework 3.0	.NET Framework 2.0 SP1, .NET Framework 2.0 SP2, .NET Framework 3.0 SP1, .NET Framework 3.0 SP2
.NET Framework 3.5	.NET Framework 3.5 SP1
.NET Framework 4	N/A

To change a project's target framework, see How to: Target a version of .NET.

How to fix violations

To remove the dependency on the service pack, remove all usages of the new member or type. If this is a

deliberate dependency, either suppress the warning or turn off this rule.

When to suppress warnings

Do not suppress a warning from this rule if this was not a deliberate dependency on the specified service pack. In this situation, your application might fail to run on systems without this service pack installed. Suppress the warning or turn off this rule if this was a deliberate dependency.

Example

The following example shows a class that uses the type DateTimeOffset that is only available in .NET 2.0 Service Pack 1. This example requires that .NET Framework 2.0 has been selected in the Target Framework drop-down list in the Project properties.

```
using System;
namespace Samples
{
    public class LibraryBook
    {
        private readonly string _Title;
        private DateTimeOffset _CheckoutDate;   // Violates this rule
        public LibraryBook(string title)
        {
            _Title = title;
        }
        public string Title
        {
        get { return _Title; }
        }
        public DateTimeOffset CheckoutDate      // Violates this rule
        {
            get { return _CheckoutDate; }
            set { _CheckoutDate = value; }
        }
    }
}
```

Example

The following example fixes the previously described violation by replacing usages of the DateTimeOffset type with the DateTime type.

```
using System;
namespace Samples
{
    public class LibraryBook
    {
        private readonly string _Title;
        private DateTime _CheckoutDate;
        public LibraryBook(string title)
        {
            _Title = title;
        }
        public string Title
        {
            get { return _Title; }
        }
        public DateTime CheckoutDate
        {
            get { return _CheckoutDate; }
            set { _CheckoutDate = value; }
        }
    }
}
```

See also

- Portability Warnings
- Framework targeting overview

Reliability warnings

8/4/2020 • 2 minutes to read • Edit Online

Reliability warnings support library and application reliability, such as correct memory and thread usage. The reliability rules include:

RULE	DESCRIPTION
CA2000: Dispose objects before losing scope	Because an exceptional event might occur that will prevent the finalizer of an object from running, the object should be explicitly disposed before all references to it are out of scope.
CA2001: Avoid calling problematic methods	A member calls a potentially dangerous or problematic method.
CA2002: Do not lock on objects with weak identity	An object is said to have a weak identity when it can be directly accessed across application domain boundaries. A thread that tries to acquire a lock on an object that has a weak identity can be blocked by a second thread in a different application domain that has a lock on the same object.
CA2003: Do not treat fibers as threads	A managed thread is being treated as a Win32 thread.
CA2004: Remove calls to GC.KeepAlive	If you are converting to SafeHandle usage, remove all calls to GC.KeepAlive (object). In this case, classes should not have to call GC.KeepAlive, assuming they do not have a finalizer but rely on SafeHandle to finalize the OS handle for them.
CA2006: Use SafeHandle to encapsulate native resources	Use of IntPtr in managed code might indicate a potential security and reliability problem. All uses of IntPtr must be reviewed to determine whether use of a SafeHandle, or similar technology, is required in its place.
CA2007: Do not directly await a Task	An asynchronous method awaits a Task directly.
CA2009: Do not call ToImmutableCollection on an ImmutableCollection value	`ToImmutable` method was unnecessarily called on an immutable collection from System.Collections.Immutable namespace.
CA2011: Do not assign property within its setter	A property was accidentally assigned a value within its own set accessor.
CA2012: Use ValueTasks correctly	ValueTasks returned from member invocations are intended to be directly awaited. Attempts to consume a ValueTask multiple times or to directly access one's result before it's known to be completed may result in an exception or corruption. Ignoring such a ValueTask is likely an indication of a functional bug and may degrade performance.

RULE	DESCRIPTION
CA2013: Do not use ReferenceEquals with value types	When comparing values using System.Object.ReferenceEquals, if objA and objB are value types, they are boxed before they are passed to the ReferenceEquals method. This means that even if both objA and objB represent the same instance of a value type, the ReferenceEquals method nevertheless returns false.
CA2014: Do not use stackalloc in loops.	Stack space allocated by a stackalloc is only released at the end of the current method's invocation. Using it in a loop can result in unbounded stack growth and eventual stack overflow conditions.
CA2015: Do not define finalizers for types derived from MemoryManager<T>	Adding a finalizer to a type derived from MemoryManager<T> may permit memory to be freed while it is still in use by a Span<T>.
CA2016: Forward the CancellationToken parameter to methods that take one	Forward the `CancellationToken` parameter to methods that take one to ensure the operation cancellation notifications gets properly propagated, or pass in `CancellationToken.None` explicitly to indicate intentionally not propagating the token.

CA2000: Dispose objects before losing scope

8/4/2020 • 6 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2000
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

A local object of an IDisposable type is created, but the object is not disposed before all references to the object are out of scope.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

If a disposable object is not explicitly disposed before all references to it are out of scope, the object will be disposed at some indeterminate time when the garbage collector runs the finalizer of the object. Because an exceptional event might occur that will prevent the finalizer of the object from running, the object should be explicitly disposed instead.

Special cases

Rule CA2000 does not fire for local objects of the following types even if the object is not disposed:

- System.IO.Stream
- System.IO.StringReader
- System.IO.TextReader
- System.IO.TextWriter
- System.Resources.IResourceReader

Passing an object of one of these types to a constructor and then assigning it to a field indicates a *dispose ownership transfer* to the newly constructed type. That is, the newly constructed type is now responsible for disposing of the object. If your code passes an object of one of these types to a constructor, no violation of rule CA2000 occurs even if the object is not disposed before all references to it are out of scope.

How to fix violations

To fix a violation of this rule, call Dispose on the object before all references to it are out of scope.

You can use the `using` statement (`Using` in Visual Basic) to wrap objects that implement IDisposable. Objects that are wrapped in this manner are automatically disposed at the end of the `using` block. However, the following situations should not or cannot be handled with a `using` statement:

- To return a disposable object, the object must constructed in a `try/finally` block outside of a `using` block.
- Do not initialize members of a disposable object in the constructor of a `using` statement.

- When constructors that are protected by only one exception handler are nested in the acquisition part of a `using` statement, a failure in the outer constructor can result in the object created by the nested constructor never being closed. In the following example, a failure in the StreamReader constructor can result in the FileStream object never being closed. CA2000 flags a violation of the rule in this case.

```
using (StreamReader sr = new StreamReader(new FileStream("C:\myfile.txt", FileMode.Create)))
{ ... }
```

- Dynamic objects should use a shadow object to implement the dispose pattern of IDisposable objects.

When to suppress warnings

Do not suppress a warning from this rule unless:

- You've called a method on your object that calls `Dispose`, such as Close
- The method that raised the warning returns an IDisposable object that wraps your object
- The allocating method does not have dispose ownership; that is, the responsibility to dispose the object is transferred to another object or wrapper that's created in the method and returned to the caller

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2000.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2000.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2000.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2000.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2000.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Related rules

- CA2213: Disposable fields should be disposed
- CA2202: Do not dispose objects multiple times

Example

If you're implementing a method that returns a disposable object, use a try/finally block without a catch block to make sure that the object is disposed. By using a try/finally block, you allow exceptions to be raised at the fault point and make sure that object is disposed.

In the OpenPort1 method, the call to open the ISerializable object SerialPort or the call to SomeMethod can fail. A CA2000 warning is raised on this implementation.

In the OpenPort2 method, two SerialPort objects are declared and set to null:

- `tempPort`, which is used to test that the method operations succeed.
- `port`, which is used for the return value of the method.

The `tempPort` is constructed and opened in a `try` block, and any other required work is performed in the same `try` block. At the end of the `try` block, the opened port is assigned to the `port` object that will be returned and the `tempPort` object is set to `null`.

The `finally` block checks the value of `tempPort`. If it is not null, an operation in the method has failed, and `tempPort` is closed to make sure that any resources are released. The returned port object will contain the opened SerialPort object if the operations of the method succeeded, or it will be null if an operation failed.

```
public SerialPort OpenPort1(string portName)
{
   SerialPort port = new SerialPort(portName);
   port.Open();  //CA2000 fires because this might throw
   SomeMethod(); //Other method operations can fail
   return port;
}

public SerialPort OpenPort2(string portName)
{
   SerialPort tempPort = null;
   SerialPort port = null;
   try
   {
      tempPort = new SerialPort(portName);
      tempPort.Open();
      SomeMethod();
      //Add any other methods above this line
      port = tempPort;
      tempPort = null;

   }
   finally
   {
      if (tempPort != null)
      {
         tempPort.Close();
      }
   }
   return port;
}
```

```
Public Function OpenPort1(ByVal PortName As String) As SerialPort

   Dim port As New SerialPort(PortName)
   port.Open()    'CA2000 fires because this might throw
   SomeMethod()   'Other method operations can fail
   Return port

End Function

Public Function OpenPort2(ByVal PortName As String) As SerialPort

   Dim tempPort As SerialPort = Nothing
   Dim port As SerialPort = Nothing

   Try
      tempPort = New SerialPort(PortName)
      tempPort.Open()
      SomeMethod()
      'Add any other methods above this line
      port = tempPort
      tempPort = Nothing

   Finally
      If Not tempPort Is Nothing Then
         tempPort.Close()
      End If

   End Try

   Return port

End Function
```

Example

By default, the Visual Basic compiler has all arithmetic operators check for overflow. Therefore, any Visual Basic arithmetic operation might throw an OverflowException. This could lead to unexpected violations in rules such as CA2000. For example, the following CreateReader1 function will produce a CA2000 violation because the Visual Basic compiler is emitting an overflow checking instruction for the addition that could throw an exception that would cause the StreamReader not to be disposed.

To fix this, you can disable the emitting of overflow checks by the Visual Basic compiler in your project or you can modify your code as in the following CreateReader2 function.

To disable the emitting of overflow checks, right-click the project name in Solution Explorer and then click **Properties**. Click **Compile**, click **Advanced Compile Options**, and then check **Remove integer overflow checks**.

```
Public Function CreateReader1(ByVal x As Integer) As StreamReader
   Dim local As New StreamReader("C:\Temp.txt")
   x += 1
   Return local
End Function


Public Function CreateReader2(ByVal x As Integer) As StreamReader
   Dim local As StreamReader = Nothing
   Dim localTemp As StreamReader = Nothing
   Try
      localTemp = New StreamReader("C:\Temp.txt")
      x += 1
      local = localTemp
      localTemp = Nothing
      Finally
         If (Not (localTemp Is Nothing)) Then
            localTemp.Dispose()
         End If
   End Try
   Return local
End Function
```

See also

- IDisposable
- Dispose Pattern

CA2001: Avoid calling problematic methods

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2001
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

A member calls a potentially dangerous or problematic method.

Rule description

Avoid making unnecessary and potentially dangerous method calls. A violation of this rule occurs when a member calls one of the following methods:

METHOD	DESCRIPTION
System.GC.Collect	Calling GC.Collect can significantly affect application performance and is rarely necessary. For more information, see Rico Mariani's Performance Tidbits blog entry on MSDN.
System.Threading.Thread.Resume System.Threading.Thread.Suspend	Thread.Suspend and Thread.Resume have been deprecated because of their unpredictable behavior. Use other classes in the System.Threading namespace, such as Monitor, Mutex, and Semaphore, to synchronize threads or protect resources.
System.Runtime.InteropServices.SafeHandle.DangerousGetHandle	The `DangerousGetHandle` method poses a security risk because it can return a handle that's not valid. For more information about how to use the `DangerousGetHandle` method safely, see the DangerousAddRef and DangerousRelease methods.
System.Reflection.Assembly.LoadFrom System.Reflection.Assembly.LoadFile System.Reflection.Assembly.LoadWithPartialName	These methods can load assemblies from unexpected locations. For example, see Suzanne Cook's .NET CLR Notes blog posts LoadFile vs. LoadFrom and Choosing a Binding Context for information about methods that load assemblies.

METHOD	DESCRIPTION
CoSetProxyBlanket CoInitializeSecurity	By the time the user code starts executing in a managed process, it's too late to reliably call `CoSetProxyBlanket`. The common language runtime (CLR) takes initialization actions that may prevent the users P/Invoke from succeeding. If you do have to call `CoSetProxyBlanket` for a managed application, we recommend that you start the process by using a native code (C++) executable, call `CoSetProxyBlanket` in the native code, and then start your managed code application in process. (Be sure to specify a runtime version number.)

How to fix violations

To fix a violation of this rule, remove or replace the call to the dangerous or problematic method.

When to suppress warnings

Suppress messages from this rule only when no alternatives to the problematic method are available.

See also

- Reliability Warnings

CA2002: Do not lock on objects with weak identity

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2002
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

A thread attempts to acquire a lock on an object that has a weak identity.

Rule description

An object is said to have a weak identity when it can be directly accessed across application domain boundaries. A thread that tries to acquire a lock on an object that has a weak identity can be blocked by a second thread in a different application domain that has a lock on the same object.

The following types have a weak identity and are flagged by the rule:

- String
- Arrays of value types, including integral types, floating-point types, and Boolean.
- MarshalByRefObject
- ExecutionEngineException
- OutOfMemoryException
- StackOverflowException
- MemberInfo
- ParameterInfo
- Thread
- this or Me object (starting in FxCop 3.0)

How to fix violations

To fix a violation of this rule, use an object from a type that is not in the list in the Description section.

When to suppress warnings

It is safe to suppress the warning if the locked object is `this` or `Me` and the visibility of the self object type is private or internal, and the instance is not accessible using any public reference.

Otherwise, do not suppress a warning from this rule.

Related rules

CA2213: Disposable fields should be disposed

Example

The following example shows some object locks that violate the rule.

```
Imports System
Imports System.IO
Imports System.Reflection
Imports System.Threading

Namespace ReliabilityLibrary

   Class WeakIdentities

      Sub SyncLockOnWeakId1()

         SyncLock GetType(WeakIdentities)
         End SyncLock

      End Sub

      Sub SyncLockOnWeakId2()

         Dim stream As New MemoryStream()
         SyncLock stream
         End SyncLock

      End Sub

      Sub SyncLockOnWeakId3()

         SyncLock "string"
         End SyncLock

      End Sub

      Sub SyncLockOnWeakId4()

         Dim member As MemberInfo = _
            Me.GetType().GetMember("SyncLockOnWeakId1")(0)
         SyncLock member
         End SyncLock

      End Sub

      Sub SyncLockOnWeakId5()

         Dim outOfMemory As New OutOfMemoryException()
         SyncLock outOfMemory
         End SyncLock

      End Sub

   End Class

End Namespace
```

```
using System;
using System.IO;
using System.Reflection;
using System.Threading;

namespace ReliabilityLibrary
{
   class WeakIdentities
   {
      void LockOnWeakId1()
      {
         lock(typeof(WeakIdentities)) {}
      }

      void LockOnWeakId2()
      {
         MemoryStream stream = new MemoryStream();
         lock(stream) {}
      }

      void LockOnWeakId3()
      {
         lock("string") {}
      }

      void LockOnWeakId4()
      {
         MemberInfo member = this.GetType().GetMember("LockOnWeakId1")[0];
         lock(member) {}
      }
      void LockOnWeakId5()
      {
         OutOfMemoryException outOfMemory = new OutOfMemoryException();
         lock(outOfMemory) {}
      }
   }
}
```

See also

- Monitor
- AppDomain
- lock Statement (C#)
- SyncLock Statement (Visual Basic)

CA2003: Do not treat fibers as threads

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2003
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

A managed thread is being treated as a Win32 thread.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Do not assume a managed thread is a Win32 thread; it's a fiber. The common language runtime (CLR) runs managed threads as fibers in the context of real threads that are owned by SQL. These threads can be shared across AppDomains and even databases in the SQL Server process. Using managed thread local storage works, but you may not use unmanaged thread local storage or assume that your code will run on the current OS thread again. Do not change settings such as the locale of the thread. Do not call CreateCriticalSection or CreateMutex via P/Invoke because they require that the thread that enters a lock must also exit the lock. Because the thread that enters a lock doesn't exit a lock when you use fibers, Win32 critical sections and mutexes are useless in SQL. You may safely use most of the state on a managed Thread object, including managed thread local storage and the current user interface (UI) culture of the thread. However, for programming model reasons, you won't be able to change the current culture of a thread when you use SQL. This limitation will be enforced through a new permission.

How to fix violations

Examine your usage of threads and change your code accordingly.

When to suppress warnings

Do not suppress this rule.

CA2004: Remove calls to GC.KeepAlive

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2004
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

Classes use `SafeHandle` but still contain calls to `GC.KeepAlive`.

Rule description

If you are converting to `SafeHandle` usage, remove all calls to `GC.KeepAlive` (object). In this case, classes should not have to call `GC.KeepAlive`,assuming they do not have a finalizer but rely on `SafeHandle` to complete the OS handle for them. Although the cost of leaving in a call to `GC.KeepAlive` might be negligible as measured by performance, the perception that a call to `GC.KeepAlive` is either necessary or sufficient to solve a lifetime issue that might no longer exist makes the code harder to maintain.

How to fix violations

Remove calls to `GC.KeepAlive`.

When to suppress warnings

You can suppress this warning only if it is not technically correct to convert to `SafeHandle` usage in your class.

CA2006: Use SafeHandle to encapsulate native resources

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2006
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

Managed code uses IntPtr to access native resources.

Rule description

Use of `IntPtr` in managed code might indicate a potential security and reliability problem. All uses of `IntPtr` must be reviewed to determine whether the use of a SafeHandle , or a similar technology, is required in its place. Problems will occur if the `IntPtr` represents some native resource, such as memory, a file handle, or a socket, that the managed code is considered to own. If the managed code owns the resource, it must also release the native resources associated with it, because a failure to do so would cause resource leakage.

In such scenarios, security or reliability problems will also exist if multithreaded access is allowed to the `IntPtr` and a way of releasing the resource that is represented by the `IntPtr` is provided. These problems involve recycling of the `IntPtr` value on resource release while simultaneous use of the resource is being made on another thread. This can cause race conditions where one thread can read or write data that is associated with the wrong resource. For example, if your type stores an OS handle as an `IntPtr` and allows users to call both **Close** and any other method that uses that handle simultaneously and without some kind of synchronization, your code has a handle recycling problem.

This handle recycling problem can cause data corruption and, frequently, a security vulnerability. `SafeHandle` and its sibling class CriticalHandle provide a mechanism to encapsulate a native handle to a resource so that such threading problems can be avoided. Additionally, you can use `SafeHandle` and its sibling class `CriticalHandle` for other threading issues, for example, to carefully control the lifetime of managed objects that contain a copy of the native handle over calls to native methods. In this situation, you can often remove calls to `GC.KeepAlive`. The performance overhead that you incur when you use `SafeHandle` and, to a lesser degree, `CriticalHandle`, can frequently be reduced through careful design.

How to fix violations

Convert `IntPtr` usage to `SafeHandle` to safely manage access to native resources. See the SafeHandle reference article for examples.

When to suppress warnings

Do not suppress this warning.

See also

- IDisposable

CA2007: Do not directly await a Task

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2007
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

An asynchronous method awaits a Task directly.

Rule description

When an asynchronous method awaits a Task directly, continuation usually occurs in the same thread that created the task, depending on the async context. This behavior can be costly in terms of performance and can result in a deadlock on the UI thread. Consider calling Task.ConfigureAwait(Boolean) to signal your intention for continuation.

This rule was introduced with FxCop analyzers and doesn't exist in legacy FxCop analysis.

How to fix violations

To fix violations, call ConfigureAwait on the awaited Task. You can pass either `true` or `false` for the `continueOnCapturedContext` parameter.

- Calling `ConfigureAwait(true)` on the task has the same behavior as not explicitly calling ConfigureAwait. By explicitly calling this method, you're letting readers know you intentionally want to perform the continuation on the original synchronization context.
- Call `ConfigureAwait(false)` on the task to schedule continuations to the thread pool, thereby avoiding a deadlock on the UI thread. Passing `false` is a good option for app-independent libraries.

When to suppress warnings

You can suppress this warning if you know that the consumer is not a graphical user interface (GUI) app or if the consumer does not have a SynchronizationContext.

Example

The following code snippet generates the warning:

```
public async Task Execute()
{
    Task task = null;
    await task;
}
```

To fix the violation, call ConfigureAwait on the awaited Task:

```
public async Task Execute()
{
    Task task = null;
    await task.ConfigureAwait(false);
}
```

Configurability

You can configure whether you want to exclude asynchronous methods that don't return a value from this rule. To exclude these kinds of methods, add the following key-value pair to an .editorconfig file in your project:

```
# Package version 2.9.0 and later
dotnet_code_quality.CA2007.exclude_async_void_methods = true

# Package version 2.6.3 and earlier
dotnet_code_quality.CA2007.skip_async_void_methods = true
```

You can also configure which output assembly kinds to apply this rule to. For example, to only apply this rule to code that produces a console application or a dynamically linked library (that is, not a UI app), add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2007.output_kind = ConsoleApplication, DynamicallyLinkedLibrary
```

For more information, see Configure FxCop analyzers.

See also

- ConfigureAwait FAQ
- Should I await a task with ConfigureAwait(false)?
- Install FxCop analyzers in Visual Studio

CA2009: Do not call ToImmutableCollection on an ImmutableCollection value

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2009
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

`ToImmutable` method was unnecessarily called on an immutable collection from System.Collections.Immutable namespace.

Rule description

System.Collections.Immutable namespace contains types that define immutable collections. This rule analyzes the following immutable collection types:

- System.Collections.Immutable.ImmutableArray<T>
- System.Collections.Immutable.ImmutableList<T>
- System.Collections.Immutable.ImmutableHashSet<T>
- System.Collections.Immutable.ImmutableSortedSet<T>
- System.Collections.Immutable.ImmutableDictionary<TKey,TValue>
- System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>

These types define extension methods that create a new immutable collection from an existing IEnumerable<T> collection.

- ImmutableArray<T> defines ToImmutableArray.
- ImmutableList<T> defines ToImmutableList.
- ImmutableHashSet<T> defines ToImmutableHashSet.
- ImmutableSortedSet<T> defines ToImmutableSortedSet.
- ImmutableDictionary<TKey,TValue> defines ToImmutableDictionary.
- ImmutableSortedDictionary<TKey,TValue> defines ToImmutableSortedDictionary.

These extension methods are designed to convert a mutable collection to an immutable collection. However, the caller might accidentally pass in an immutable collection as input to these methods. This can represent a performance and/or a functional issue.

- Performance issue: Unnecessary creation of a duplicate immutable collection. The original collection was already immutable and can be used directly.
- Potential functional issue: Caller assumed to be operating on a mutable collection, when it actually had an immutable collection.

> **NOTE**
>
> Rule CA2009 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.6.

How to fix violations

To fix violations, remove the redundant `ToImmutable` call on an immutable collection. For example, the following two code snippets show a violation of the rule and how to fix them:

```
using System;
using System.Collections.Generic;
using System.Collections.Immutable;

public class C
{
    public void M(IEnumerable<int> collection, ImmutableArray<int> immutableArray)
    {
        // This is fine.
        M2(collection.ToImmutableArray());

        // This leads to CA2009.
        M2(immutableArray.ToImmutableArray());
    }

    private void M2(ImmutableArray<int> immutableArray)
    {
        Console.WriteLine(immutableArray.Length);
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Collections.Immutable;

public class C
{
    public void M(IEnumerable<int> collection, ImmutableArray<int> immutableArray)
    {
        // This is fine.
        M2(collection.ToImmutableArray());

        // This is now fine.
        M2(immutableArray);
    }

    private void M2(ImmutableArray<int> immutableArray)
    {
        Console.WriteLine(immutableArray.Length);
    }
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the violation and press **Ctrl+.** (period). Choose **Remove redundant call** from the list of options that's presented.

When to suppress warnings

Do not suppress violations from this rule, unless you're not concerned about the performance impact from unnecessary allocations of immutable collections.

See also

- Reliability warnings
- Performance warnings

CA2011: Do not assign property within its setter

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2011
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

A property was accidentally assigned a value within its own set accessor.

Rule description

Assigning a property to itself in its set accessor leads to an infinite chain of recursive calls to the set accessor. This results in a StackOverflowException at runtime. Such a mistake is common when the property and the backing field to store the property value have similar names. Instead of assigning the value to the backing field, it was accidentally assigned to the property itself.

> **NOTE**
> Rule CA2011 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.6.

How to fix violations

To fix violations, replace the violating assignment to the property with either an assignment to the backing field or switch to using an auto-property. For example, the following code snippet shows a violation of the rule and a couple of ways to fix it:

```
public class C
{
    // Backing field for property 'P'
    private int p;

    public int P
    {
        get
        {
            return p;
        }
        set
        {
            // CA2011: Accidentally assigned to property, instead of the backing field.
            P = value;
        }
    }
}
```

```
public class C
{
    // Backing field for property 'P'
    private int _p;

    public int P
    {
        get
        {
            return _p;
        }
        set
        {
            // Option 1: Assign to backing field and rename the backing field for clarity.
            _p = value;
        }
    }
}
```

```
public class C
{
    // Option 2: Use auto-property.
    public int P { get; set; }
}
```

When to suppress warnings

It is fine to suppress violations from this rule if you are sure that the recursive calls to the set accessor are conditionally guarded to prevent infinite recursion.

Related rules

- CA2245: Do not assign a property to itself

See also

- Reliability warnings

CA2012: Use ValueTasks correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2012
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

A ValueTask instance returned from a member invocation is used in a way that could lead to exceptions, corruption, or poor performance.

Rule description

ValueTask instances returned from member invocations are intended to be directly awaited. Attempts to consume a ValueTask multiple times or to directly access one's result before it's known to be completed may result in an exception or corruption. Ignoring such a ValueTask is likely an indication of a functional bug and may degrade performance.

How to fix violations

In general, ValueTasks should be directly awaited rather than discarded or stored into other locations like local variables or fields.

When to suppress warnings

For ValueTasks returned from arbitrary member calls, the caller needs to assume that the ValueTask must be consumed (e.g. awaited) once and only once. However, if the developer also controls the member being invoked and has complete knowledge of its implementation, the developer may know it's safe to suppress the warning, for example, if the return ValueTask always wraps a Task object.

See also

- Reliability warnings

CA2013: Do not use ReferenceEquals with value types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2013
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

Using System.Object.ReferenceEquals method to test one or more value types for equality.

Rule description

When comparing values using ReferenceEquals, if objA and objB are value types, they are boxed before they are passed to the ReferenceEquals method. This means that even if both objA and objB represent the same instance of a value type, the ReferenceEquals method nevertheless returns false, as the following example shows.

How to fix violations

To fix the violation, replace it with a more appropriate equality check such as `==`.

```
int int1 = 1, int2 = 1;

// Violation occurs, returns false.
Console.WriteLine(Object.ReferenceEquals(int1, int2));  // false

// Use appropriate equality operator or method instead
Console.WriteLine(int1 == int2);                        // true
Console.WriteLine(Object.Equals(int1, int2));           // true
```

When to suppress warnings

It is NOT safe to suppress a warning from this rule, we recommend using the more appropriate equality operator such as `==`.

Related rules

- CA2231: Overload operator equals on overriding ValueType.Equals
- CA2224: Override equals on overloading operator equals
- CA2218: Override GetHashCode on overriding Equals

See also

- Reliability warnings

CA2014: Do not use stackalloc in loops

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2014
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

Using the C# stackalloc keyword inside of a loop.

Rule description

The C# stackalloc keyword allocates memory from the current stack frame, and that memory may not be released until the current method call returns. If the stackalloc keyword is used in a loop, it can lead to stack overflows due to exhausting the stack memory.

How to fix violations

Move the stackalloc to outside of all loops in the method.

When to suppress warnings

It may be safe to suppress the warning when the containing loop or loops are invoked only a finite number of times, such that the overall amount of memory allocated across all stackalloc operations is known to be relatively small.

See also

- Reliability warnings

CA2015: Do not define finalizers for types derived from MemoryManager<T>

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2015
Category	Microsoft.Reliability
Breaking change	Non-breaking

Cause

Defining finalizers for types derived from MemoryManager<T>

Rule description

Adding a finalizer to a type derived from MemoryManager<T> is likely an indication of a bug, as it suggests a native resource that could have been handed out in a Span<T> is getting cleaned up and potentially while it is still in use by the Span<T>.

> **NOTE**
> The MemoryManager<T> class is intended for advanced scenarios. Most developers do not need to use it.

How to fix violations

To fix the violation, remove the finalizer definition.

```
class DerivedClass <T> : MemoryManager<T>
{
    public override bool Dispose(bool disposing)
    {
        if (disposing)
        {
            _handle.Dispose();
        }
    }

    ...

    // Violation occurs, remove the finalizer to fix the warning.
    ~DerivedClass() => Dispose(false);
}
```

When to suppress warnings

It is fine to suppress a violation of this rule if the intent is to create a finalizer for debugging or validation purposes.

Related rules

- CA1821: Remove empty finalizers

See also

- Reliability warnings

CA2016: Forward the CancellationToken parameter to methods that take one

8/4/2020 • 3 minutes to read • Edit Online

TypeName	ForwardCancellationTokenToInvocations
CheckId	CA2016
Category	Microsoft.Performance
Breaking Change	Non-breaking

Cause

This rule locates method invocations that could accept a CancellationToken parameter, but are not passing any, and suggests to forward the parent method's `CancellationToken` to them.

Rule description

This rule analyzes method definitions that take a `CancellationToken` as their last parameter, then analyzes all methods invoked in its body. If any of the method invocations can either accept a `CancellationToken` as the last parameter, or have an overload that takes a `CancellationToken` as the last parameter, then the rule suggests using that option instead to ensure that the cancellation notification gets propagated to all operations that can listen to it.

> **NOTE**
> Rule CA2016 is available in all .NET versions where the `CancellationToken` type is available. See CancellationToken "Applies to" section

How to fix violations

You can either fix them manually, or you can opt to let Visual Studio do it for you, by hovering over the light bulb that shows up next to the method invocation, and selecting the suggested change.

The following example shows two suggested changes:

It's safe to suppress a violation of this rule if you're not concerned about forwarding the canceled operation notification to lower method invocations. You can also explicitly pass `default` in C# (`Nothing` in Visual Basic) or None to suppress the rule violation.

The rule can detect a variety of violations. The following examples show cases that the rule can detect:

Example 1

The rule will suggest forwarding the `c` parameter from `MyMethod` to the `MyMethodWithDefault` invocation, because the method defines an optional token parameter:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken ct = default)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault();
        }
    }
}
```

Fix

Forward the `c` parameter:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault(c);
        }
```

If you are not concerned about forwarding cancellation notifications to lower invocations, you can either:

Explicitly pass `default`:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault(default);
        }
```

Or explicitly pass `CancellationToken.None`:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault(CancellationToken.None);
        }
```

Example 2

The rule will suggest forwarding the `c` parameter from `MyMethod` to the `MyMethodWithDefault` invocation, because the method has an overload that takes a `CancellationToken` parameter:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithOverload()
        {
        }

        public static void MyMethodWithOverload(CancellationToken ct = default)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload();
        }
    }
}
```

Fix

Forward the `c` parameter:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload(c);
        }
```

If you are not concerned about forwarding cancellation notifications to lower invocations, you can either:

Explicitly pass `default`:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload(default);
        }
```

Or explicitly pass `CancellationToken.None`:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload(CancellationToken.None);
        }
```

No diagnosis

Example 1

The `CancellationToken` parameter in the parent method is not in the last position:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken ct = default)
        {
        }

        public static void MyMethod(CancellationToken c, int lastParameter)
        {
            MyMethodWithDefault();
        }
    }
}
```

Example 2

The `CancellationToken` parameter in the default method is not in the last position:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken ct = default, int lastParameter = 0)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault();
        }
    }
}
```

Example 3

The `CancellationToken` parameter in the overload method is not in the last position:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithOverload(int lastParameter)
        {
        }
        public static void MyMethodWithOverload(CancellationToken ct, int lastParameter)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload();
        }
    }
}
```

Example 4

The parent method defines more than one `CancellationToken` parameter:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken ct = default)
        {
        }

        public static void MyMethod(CancellationToken c1, CancellationToken c2)
        {
            MyMethodWithDefault();
        }
    }
}
```

Example 5

The method with defaults defines more than one `CancellationToken` parameter:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken c1 = default, CancellationToken c2 = default)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault();
        }
    }
}
```

Example 6

The method overload defines more than one `CancellationToken` parameter:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithOverload(CancellationToken c1, CancellationToken c2)
        {
        }

        public static void MyMethodWithOverload()
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload();
        }
    }
}
```

Security warnings

8/4/2020 • 27 minutes to read • Edit Online

Security warnings support safer libraries and applications. These warnings help prevent security flaws in your program. If you disable any of these warnings, you should clearly mark the reason in code and also inform the designated security officer for your development project.

In this section

RULE	DESCRIPTION
CA2100: Review SQL queries for security vulnerabilities	A method sets the System.Data.IDbCommand.CommandText property by using a string that is built from a string argument to the method. This rule assumes that the string argument contains user input. A SQL command string built from user input is vulnerable to SQL injection attacks.
CA2102: Catch non-CLSCompliant exceptions in general handlers	A member in an assembly that is not marked with the RuntimeCompatibilityAttribute or is marked RuntimeCompatibility(WrapNonExceptionThrows = false) contains a catch block that handles System.Exception and does not contain an immediately following general catch block.
CA2103: Review imperative security	A method uses imperative security and might be constructing the permission by using state information or return values that can change while the demand is active. Use declarative security whenever possible.
CA2104: Do not declare read only mutable reference types	An externally visible type contains an externally visible read-only field that is a mutable reference type. A mutable type is a type whose instance data can be modified.
CA2105: Array fields should not be read only	When you apply the read-only (ReadOnly in Visual Basic) modifier to a field that contains an array, the field cannot be changed to reference a different array. However, the elements of the array stored in a read-only field can be changed.
CA2106: Secure asserts	A method asserts a permission and no security checks are performed on the caller. Asserting a security permission without performing any security checks can leave an exploitable security weakness in your code.
CA2107: Review deny and permit only usage	Using the PermitOnly method and CodeAccessPermission.Deny security actions should be used only by those with an advanced knowledge of .NET security. Code that uses these security actions should undergo a security review.
CA2108: Review declarative security on value types	A public or protected value type is secured by Data Access or Link Demands.
CA2109: Review visible event handlers	A public or protected event-handling method was detected. Event-handling methods should not be exposed unless absolutely necessary.
CA2111: Pointers should not be visible	A pointer is not private, internal, or read-only. Malicious code can change the value of the pointer, potentially allowing access to arbitrary locations in memory or causing application or system failures.

RULE	DESCRIPTION
CA2112: Secured types should not expose fields	A public or protected type contains public fields and is secured by Link Demands. If code has access to an instance of a type that is secured by a link demand, the code does not have to satisfy the link demand to access the type's fields.
CA2114: Method security should be a superset of type	A method should not have both method-level and type-level declarative security for the same action.
CA2115: Call GC.KeepAlive when using native resources	This rule detects errors that might occur because an unmanaged resource is being finalized while it is still being used in unmanaged code.
CA2116: APTCA methods should only call APTCA methods	When the APTCA (AllowPartiallyTrustedCallers) attribute is present on a fully trusted assembly, and the assembly executes code in another assembly that does not allow partially trusted callers, a security exploit is possible.
CA2117: APTCA types should only extend APTCA base types	When the APTCA (AllowPartiallyTrustedCallers) attribute is present on a fully trusted assembly, and a type in the assembly inherits from a type that does not allow partially trusted callers, a security exploit is possible.
CA2118: Review SuppressUnmanagedCodeSecurityAttribute usage	SuppressUnmanagedCodeSecurityAttribute changes the default security system behavior for members that execute unmanaged code that uses COM interop or platform invocation. This attribute is primarily used to increase performance; however, the performance gains come with significant security risks.
CA2119: Seal methods that satisfy private interfaces	An inheritable public type provides an overridable method implementation of an internal (Friend in Visual Basic) interface. To fix a violation of this rule, prevent the method from being overridden outside the assembly.
CA2120: Secure serialization constructors	This type has a constructor that takes a System.Runtime.Serialization.SerializationInfo object and a System.Runtime.Serialization.StreamingContext object (the signature of the serialization constructor). This constructor is not secured by a security check, but one or more of the regular constructors in the type are secured.
CA2121: Static constructors should be private	The system calls the static constructor before the first instance of the type is created or any static members are referenced. If a static constructor is not private, it can be called by code other than the system. Depending on the operations that are performed in the constructor, this can cause unexpected behavior.
CA2122: Do not indirectly expose methods with link demands	A public or protected member has Link Demands and is called by a member that does not perform any security checks. A link demand checks the permissions of the immediate caller only.
CA2123: Override link demands should be identical to base	This rule matches a method to its base method, which is either an interface or a virtual method in another type, and then compares the link demands on each. If this rule is violated, a malicious caller can bypass the link demand just by calling the unsecured method.
CA2124: Wrap vulnerable finally clauses in outer try	A public or protected method contains a try/finally block. The finally block appears to reset the security state and is not itself enclosed in a finally block.
CA2126: Type link demands require inheritance demands	A public unsealed type is protected with a link demand and has an overridable method. Neither the type nor the method is protected with an inheritance demand.

RULE	DESCRIPTION
CA2130: Security critical constants should be transparent	Transparency enforcement is not enforced for constant values because compilers inline constant values so that no lookup is required at run time. Constant fields should be security transparent so that code reviewers do not assume that transparent code cannot access the constant.
CA2131: Security critical types may not participate in type equivalence	A type participates in type equivalence and either the type itself, or a member or field of the type, is marked with the SecurityCriticalAttribute attribute. This rule fires on any critical types or types that contain critical methods or fields that are participating in type equivalence. When the CLR detects such a type, it fails to load it with a TypeLoadException at run time. Typically, this rule fires only when users implement type equivalence manually rather than by relying on tlbimp and the compilers to do the type equivalence.
CA2132: Default constructors must be at least as critical as base type default constructors	Types and members that have the SecurityCriticalAttribute cannot be used by Silverlight application code. Security-critical types and members can be used only by trusted code in the .NET Framework for Silverlight class library. Because a public or protected construction in a derived class must have the same or greater transparency than its base class, a class in an application cannot be derived from a class marked SecurityCritical.
CA2133: Delegates must bind to methods with consistent transparency	This warning fires on a method that binds a delegate that is marked with the SecurityCriticalAttribute to a method that is transparent or that is marked with the SecuritySafeCriticalAttribute. The warning also fires a method that binds a delegate that is transparent or safe-critical to a critical method.
CA2134: Methods must keep consistent transparency when overriding base methods	This rule fires when a method marked with the SecurityCriticalAttribute overrides a method that is transparent or marked with the SecuritySafeCriticalAttribute. The rule also fires when a method that is transparent or marked with the SecuritySafeCriticalAttribute overrides a method that is marked with a SecurityCriticalAttribute. The rule is applied when overriding a virtual method or implementing an interface.
CA2135: Level 2 assemblies should not contain LinkDemands	LinkDemands are deprecated in the level 2 security rule set. Instead of using LinkDemands to enforce security at just-in-time (JIT) compilation time, mark the methods, types, and fields with the SecurityCriticalAttribute attribute.
CA2136: Members should not have conflicting transparency annotations	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.
CA2137: Transparent methods must contain only verifiable IL	A method contains unverifiable code or returns a type by reference. This rule fires on attempts by security transparent code to execute unverifiable MSIL (Microsoft Intermediate Language). However, the rule does not contain a full IL verifier, and instead uses heuristics to catch most violations of MSIL verification.
CA2138: Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute	A security transparent method calls a method that is marked with the SuppressUnmanagedCodeSecurityAttribute attribute.

RULE	DESCRIPTION
CA2139: Transparent methods may not use the HandleProcessCorruptingExceptions attribute	This rule fires on any method that's transparent and attempts to handle a process corrupting exception by using the HandleProcessCorruptedStateExceptionsAttribute attribute. A process corrupting exception is a CLR version 4.0 exception classification of exceptions such as AccessViolationException. The HandleProcessCorruptedStateExceptionsAttribute attribute may only be used by security critical methods, and will be ignored if it is applied to a transparent method.
CA2140: Transparent code must not reference security critical items	Methods that are marked with SecurityTransparentAttribute call non-public members that are marked as SecurityCritical. This rule analyzes all methods and types in an assembly that is mixed transparent and critical, and flags any calls from transparent code to non-public critical code that are not marked SecurityTreatAsSafe.
CA2141:Transparent methods must not satisfy LinkDemands	A security transparent method calls a method in an assembly that is not marked with the AllowPartiallyTrustedCallersAttribute (APTCA) attribute, or a security transparent method satisfies a LinkDemand for a type or a method.
CA2142: Transparent code should not be protected with LinkDemands	This rule fires on transparent methods which require LinkDemands to access them. Security transparent code should not be responsible for verifying the security of an operation, and therefore should not demand permissions.
CA2143: Transparent methods should not use security demands	Security transparent code should not be responsible for verifying the security of an operation, and therefore should not demand permissions. Security transparent code should use full demands to make security decisions and safe-critical code should not rely on transparent code to have made the full demand.
CA2144: Transparent code should not load assemblies from byte arrays	The security review for transparent code is not as thorough as the security review for critical code, because transparent code cannot perform security sensitive actions. Assemblies loaded from a byte array might not be noticed in transparent code, and that byte array might contain critical, or more importantly safe-critical code, that does need to be audited.
CA2145: Transparent methods should not be decorated with the SuppressUnmanagedCodeSecurityAttribute	Methods decorated with the SuppressUnmanagedCodeSecurityAttribute attribute have an implicit LinkDemand placed upon any method that calls it. This LinkDemand requires that the calling code be security critical. Marking the method that uses SuppressUnmanagedCodeSecurity with the SecurityCriticalAttribute attribute makes this requirement more obvious for callers of the method.
CA2146: Types must be at least as critical as their base types and interfaces	This rule fires when a derived type has a security transparency attribute that is not as critical as its base type or implemented interface. Only critical types can derive from critical base types or implement critical interfaces, and only critical or safe-critical types can derive from safe-critical base types or implement safe-critical interfaces.
CA2147: Transparent methods may not use security asserts	This rule analyzes all methods and types in an assembly that is either 100% transparent or mixed transparent/critical, and flags any declarative or imperative use of Assert.
CA2149: Transparent methods must not call into native code	This rule fires on any transparent method that calls directly into native code, for example, through a P/Invoke. Violations of this rule lead to a MethodAccessException in the level 2 transparency model, and a full demand for UnmanagedCode in the level 1 transparency model.

RULE	DESCRIPTION
CA2151: Fields with critical types should be security critical	To use security critical types, the code that references the type must be either security critical or security safe critical. This is true even if the reference is indirect. Therefore, having a security transparent or security safe critical field is misleading because transparent code will still be unable to access the field.
CA2153: Avoid Handling Corrupted State Exceptions	Corrupted State Exceptions (CSE) indicate that memory corruption exists in your process. Catching these rather than allowing the process to crash can lead to security vulnerabilities if an attacker can place an exploit into the corrupted memory region.
CA2300: Do not use insecure deserializer BinaryFormatter	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2301: Do not call BinaryFormatter.Deserialize without first setting BinaryFormatter.Binder	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2302: Ensure BinaryFormatter.Binder is set before calling BinaryFormatter.Deserialize	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2305: Do not use insecure deserializer LosFormatter	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2310: Do not use insecure deserializer NetDataContractSerializer	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2311: Do not deserialize without first setting NetDataContractSerializer.Binder	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2312: Ensure NetDataContractSerializer.Binder is set before deserializing	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2315: Do not use insecure deserializer ObjectStateFormatter	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2321: Do not deserialize with JavaScriptSerializer using a SimpleTypeResolver	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2322: Ensure JavaScriptSerializer is not initialized with SimpleTypeResolver before deserializing	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2326: Do not use TypeNameHandling values other than None	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2327: Do not use insecure JsonSerializerSettings	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2328: Ensure that JsonSerializerSettings are secure	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.

RULE	DESCRIPTION
CA2329: Do not deserialize with JsonSerializer using an insecure configuration	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2330: Ensure that JsonSerializer has a secure configuration when deserializing	Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects.
CA2350: Ensure DataTable.ReadXml()'s input is trusted	When deserializing a DataTable with untrusted input, an attacker can craft malicious input to perform a denial of service attack. There may be unknown remote code execution vulnerabilities.
CA2351: Ensure DataSet.ReadXml()'s input is trusted	When deserializing a DataSet with untrusted input, an attacker can craft malicious input to perform a denial of service attack. There may be unknown remote code execution vulnerabilities.
CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks	A class or struct marked with SerializableAttribute contains a DataSet or DataTable field or property, and doesn't have a GeneratedCodeAttribute.
CA2353: Unsafe DataSet or DataTable in serializable type	A class or struct marked with an XML serialization attribute or a data contract attribute contains a DataSet or DataTable field or property.
CA2354: Unsafe DataSet or DataTable in deserialized object graph can be vulnerable to remote code execution attack	Deserializing with an System.Runtime.Serialization.IFormatter serialized, and the casted type's object graph can include a DataSet or DataTable.
CA2355: Unsafe DataSet or DataTable in deserialized object graph	Deserializing when the casted or specified type's object graph can include a DataSet or DataTable.
CA2356: Unsafe DataSet or DataTable in web deserialized object graph	A method with a System.Web.Services.WebMethodAttribute or System.ServiceModel.OperationContractAttribute has a parameter that may reference a DataSet or DataTable.
CA3001: Review code for SQL injection vulnerabilities	When working with untrusted input and SQL commands, be mindful of SQL injection attacks. An SQL injection attack can execute malicious SQL commands, compromising the security and integrity of your application.
CA3002: Review code for XSS vulnerabilities	When working with untrusted input from web requests, be mindful of cross-site scripting (XSS) attacks. An XSS attack injects untrusted input into raw HTML output, allowing the attacker to execute malicious scripts or maliciously modify content in your web page.
CA3003: Review code for file path injection vulnerabilities	When working with untrusted input from web requests, be mindful of using user-controlled input when specifying paths to files.
CA3004: Review code for information disclosure vulnerabilities	Disclosing exception information gives attackers insight into the internals of your application, which can help attackers find other vulnerabilities to exploit.
CA3006: Review code for process command injection vulnerabilities	When working with untrusted input, be mindful of command injection attacks. A command injection attack can execute malicious commands on the underlying operating system, compromising the security and integrity of your server.
CA3007: Review code for open redirect vulnerabilities	When working with untrusted input, be mindful of open redirect vulnerabilities. An attacker can exploit an open redirect vulnerability to use your website to give the appearance of a legitimate URL, but redirect an unsuspecting visitor to a phishing or other malicious webpage.

RULE	DESCRIPTION
CA3008: Review code for XPath injection vulnerabilities	When working with untrusted input, be mindful of XPath injection attacks. Constructing XPath queries using untrusted input may allow an attacker to maliciously manipulate the query to return an unintended result, and possibly disclose the contents of the queried XML.
CA3009: Review code for XML injection vulnerabilities	When working with untrusted input, be mindful of XML injection attacks.
CA3010: Review code for XAML injection vulnerabilities	When working with untrusted input, be mindful of XAML injection attacks. XAML is a markup language that directly represents object instantiation and execution. That means elements created in XAML can interact with system resources (for example, network access and file system IO).
CA3011: Review code for DLL injection vulnerabilities	When working with untrusted input, be mindful of loading untrusted code. If your web application loads untrusted code, an attacker may be able to inject malicious DLLs into your process and execute malicious code.
CA3012: Review code for regex injection vulnerabilities	When working with untrusted input, be mindful of regex injection attacks. An attacker can use regex injection to maliciously modify a regular expression, to make the regex match unintended results, or to make the regex consume excessive CPU resulting in a Denial of Service attack.
CA3061: Do not add schema by URL	Do not use the unsafe overload of the Add method because it may cause dangerous external references.
CA3075: Insecure DTD Processing	If you use insecure DTDProcessing instances or reference external entity sources, the parser may accept untrusted input and disclose sensitive information to attackers.
CA3076: Insecure XSLT Script Execution	If you execute Extensible Stylesheet Language Transformations (XSLT) in .NET applications insecurely, the processor may resolve untrusted URI references that could disclose sensitive information to attackers, leading to Denial of Service and Cross-Site attacks.
CA3077: Insecure Processing in API Design, XML Document and XML Text Reader	When designing an API derived from XMLDocument and XMLTextReader, be mindful of DtdProcessing. Using insecure DTDProcessing instances when referencing or resolving external entity sources or setting insecure values in the XML may lead to information disclosure.
CA3147: Mark verb handlers with ValidateAntiForgeryToken	When designing an ASP.NET MVC controller, be mindful of cross-site request forgery attacks. A cross-site request forgery attack can send malicious requests from an authenticated user to your ASP.NET MVC controller.
CA5122 P/Invoke declarations should not be safe critical	Methods are marked as SecuritySafeCritical when they perform a security sensitive operation, but are also safe to be used by transparent code. Transparent code may never directly call native code through a P/Invoke. Therefore, marking a P/Invoke as security safe critical will not enable transparent code to call it, and is misleading for security analysis.
CA5359: Do not disable certificate validation	A certificate can help authenticate the identity of the server. Clients should validate the server certificate to ensure requests are sent to the intended server. If the ServerCertificateValidationCallback always returns `true`, any certificate will pass validation.

RULE	DESCRIPTION
CA5360: Do not call dangerous methods in deserialization	Insecure deserialization is a vulnerability which occurs when untrusted data is used to abuse the logic of an application, inflict a Denial-of-Service (DoS) attack, or even execute arbitrary code upon it being deserialized. It's frequently possible for malicious users to abuse these deserialization features when the application is deserializing untrusted data which is under their control. Specifically, invoke dangerous methods in the process of deserialization. Successful insecure deserialization attacks could allow an attacker to carry out attacks such as DoS attacks, authentication bypasses, and remote code execution.
CA5361: Do not disable SChannel use of strong crypto	Setting `Switch.System.Net.DontEnableSchUseStrongCrypto` to `true` weakens the cryptography used in outgoing Transport Layer Security (TLS) connections. Weaker cryptography can compromise the confidentiality of communication between your application and the server, making it easier for attackers to eavesdrop sensitive data.
CA5362: Potential reference cycle in deserialized object graph	If deserializing untrusted data, then any code processing the deserialized object graph needs to handle reference cycles without going into infinite loops. This includes both code that's part of a deserialization callback and code that processes the object graph after deserialization completed. Otherwise, an attacker could perform a Denial-of-Service attack with malicious data containing a reference cycle.
CA5363: Do not disable request validation	Request validation is a feature in ASP.NET that examines HTTP requests and determines whether they contain potentially dangerous content that can lead to injection attacks, including cross-site-scripting.
CA5364: Do not use deprecated security protocols	Transport Layer Security (TLS) secures communication between computers, most commonly with Hypertext Transfer Protocol Secure (HTTPS). Older protocol versions of TLS are less secure than TLS 1.2 and TLS 1.3 and are more likely to have new vulnerabilities. Avoid older protocol versions to minimize risk.
CA5365: Do Not Disable HTTP Header Checking	HTTP header checking enables encoding of the carriage return and newline characters, \r and \n, that are found in response headers. This encoding can help to avoid injection attacks that exploit an application that echoes untrusted data contained by the header.
CA5366: Use XmlReader For DataSet Read XML	Using a DataSet to read XML with untrusted data may load dangerous external references, which should be restricted by using an XmlReader with a secure resolver or with DTD processing disabled.
CA5367: Do Not Serialize Types With Pointer Fields	This rule checks whether there's a serializable class with a pointer field or property. Members that can't be serialized can be a pointer, such as static members or fields marked with NonSerializedAttribute.
CA5368: Set ViewStateUserKey For Classes Derived From Page	Setting the ViewStateUserKey property can help you prevent attacks on your application by allowing you to assign an identifier to the view-state variable for individual users so that attackers cannot use the variable to generate an attack. Otherwise, there will be vulnerabilities to cross-site request forgery.
CA5369: Use XmlReader for Deserialize	Processing untrusted DTD and XML schemas may enable loading dangerous external references, which should be restricted by using an XmlReader with a secure resolver or with DTD and XML inline schema processing disabled.
CA5370: Use XmlReader for validating reader	Processing untrusted DTD and XML schemas may enable loading dangerous external references. This dangerous loading can be restricted by using an XmlReader with a secure resolver or with DTD and XML inline schema processing disabled.

RULE	DESCRIPTION
CA5371: Use XmlReader for schema read	Processing untrusted DTD and XML schemas may enable loading dangerous external references. Using an XmlReader with a secure resolver or with DTD and XML inline schema processing disabled restricts this.
CA5372: Use XmlReader for XPathDocument	Processing XML from untrusted data may load dangerous external references, which can be restricted by using an XmlReader with a secure resolver or with DTD processing disabled.
CA5373: Do not use obsolete key derivation function	This rule detects the invocation of weak key derivation methods System.Security.Cryptography.PasswordDeriveBytes and `Rfc2898DeriveBytes.CryptDeriveKey`. System.Security.Cryptography.PasswordDeriveBytes used a weak algorithm PBKDF1.
CA5374: Do Not Use XslTransform	This rule checks if System.Xml.Xsl.XslTransform is instantiated in the code. System.Xml.Xsl.XslTransform is now obsolete and shouldn't be used.
CA5375: Do not use account shared access signature	An account SAS can delegate access to read, write, and delete operations on blob containers, tables, queues, and file shares that are not permitted with a service SAS. However, it doesn't support container-level policies and has less flexibility and control over the permissions that are granted. Once malicious users get it, your storage account will be compromised easily.
CA5376: Use SharedAccessProtocol HttpsOnly	SAS is sensitive data that can't be transported in plain text on HTTP.
CA5377: Use container level access policy	A container-level access policy can be modified or revoked at any time. It provides greater flexibility and control over the permissions that are granted.
CA5378: Do not disable ServicePointManagerSecurityProtocols	Setting `Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityPro` to `true` limits Windows Communication Framework's (WCF) Transport Layer Security (TLS) connections to using TLS 1.0. That version of TLS will be deprecated.
CA5379: Do not use weak key derivation function algorithm	The Rfc2898DeriveBytes class defaults to using the SHA1 algorithm. You should specify the hash algorithm to use in some overloads of the constructor with SHA256 or higher. Note, HashAlgorithm property only has a `get` accessor and doesn't have a `overriden` modifier.
CA5380: Do not add certificates to root store	This rule detects code that adds a certificate into the Trusted Root Certification Authorities certificate store. By default, the Trusted Root Certification Authorities certificate store is configured with a set of public CAs that has met the requirements of the Microsoft Root Certificate Program.
CA5381: Ensure certificates are not added to root store	This rule detects code that potentially adds a certificate into the Trusted Root Certification Authorities certificate store. By default, the Trusted Root Certification Authorities certificate store is configured with a set of public certification authorities (CAs) that has met the requirements of the Microsoft Root Certificate Program.
CA5382: Use secure cookies in ASP.NET Core	Applications available over HTTPS must use secure cookies, which indicate to the browser that the cookie should only be transmitted using Transport Layer Security (TLS).
CA5383: Ensure use secure cookies in ASP.NET Core	Applications available over HTTPS must use secure cookies, which indicate to the browser that the cookie should only be transmitted using Transport Layer Security (TLS).

RULE	DESCRIPTION
CA5384: Do not use digital signature algorithm (DSA)	DSA is a weak asymmetric encryption algorithm.
CA5385: Use Rivest–Shamir–Adleman (RSA) algorithm with sufficient key size	An RSA key smaller than 2048 bits is more vulnerable to brute force attacks.
CA5386: Avoid hardcoding SecurityProtocolType value	Transport Layer Security (TLS) secures communication between computers, most commonly with Hypertext Transfer Protocol Secure (HTTPS). Protocol versions TLS 1.0 and TLS 1.1 are deprecated, while TLS 1.2 and TLS 1.3 are current. In the future, TLS 1.2 and TLS 1.3 may be deprecated. To ensure that your application remains secure, avoid hardcoding a protocol version and target at least .NET Framework v4.7.1.
CA5387: Do not use weak key derivation function with insufficient iteration count	This rule checks if a cryptographic key was generated by Rfc2898DeriveBytes with an iteration count of less than 100,000. A higher iteration count can help mitigate against dictionary attacks that try to guess the generated cryptographic key.
CA5388: Ensure sufficient iteration count when using weak key derivation function	This rule checks if a cryptographic key was generated by Rfc2898DeriveBytes with an iteration count that may be less than 100,000. A higher iteration count can help mitigate against dictionary attacks that try to guess the generated cryptographic key.
CA5389: Do not add archive item's path to the target file system path	File path can be relative and can lead to file system access outside of the expected file system target path, leading to malicious config changes and remote code execution via lay-and-wait technique.
CA5390: Do not hard-code encryption key	For a symmetric algorithm to be successful, the secret key must be known only to the sender and the receiver. When a key is hard-coded, it is easily discovered. Even with compiled binaries, it is easy for malicious users to extract it. Once the private key is compromised, the cipher text can be decrypted directly and is not protected anymore.
CA5391: Use antiforgery tokens in ASP.NET Core MVC controllers	Handling a `POST`, `PUT`, `PATCH`, or `DELETE` request without validating an antiforgery token may be vulnerable to cross-site request forgery attacks. A cross-site request forgery attack can send malicious requests from an authenticated user to your ASP.NET Core MVC controller.
CA5392: Use DefaultDllImportSearchPaths attribute for P/Invokes	By default, P/Invoke functions using DllImportAttribute probe a number of directories, including the current working directory for the library to load. This can be a security issue for certain applications, leading to DLL hijacking.
CA5393: Do not use unsafe DllImportSearchPath value	There could be a malicious DLL in the default DLL search directories and assembly directories. Or, depending on where your application is run from, there could be a malicious DLL in the application's directory.
CA5394: Do not use insecure randomness	Using a cryptographically weak pseudo-random number generator may allow an attacker to predict what security-sensitive value will be generated.
CA5395: Miss HttpVerb attribute for action methods	All the action methods that create, edit, delete, or otherwise modify data needs to be protected with the antiforgery attribute from cross-site request forgery attacks. Performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.
CA5396: Set HttpOnly to true for HttpCookie	As a defense in depth measure, ensure security sensitive HTTP cookies are marked as HttpOnly. This indicates web browsers should disallow scripts from accessing the cookies. Injected malicious scripts are a common way of stealing cookies.

RULE	DESCRIPTION
CA5397: Do not use deprecated SslProtocols values	Transport Layer Security (TLS) secures communication between computers, most commonly with Hypertext Transfer Protocol Secure (HTTPS). Older protocol versions of TLS are less secure than TLS 1.2 and TLS 1.3 and are more likely to have new vulnerabilities. Avoid older protocol versions to minimize risk.
CA5398: Avoid hardcoded SslProtocols values	Transport Layer Security (TLS) secures communication between computers, most commonly with Hypertext Transfer Protocol Secure (HTTPS). Protocol versions TLS 1.0 and TLS 1.1 are deprecated, while TLS 1.2 and TLS 1.3 are current. In the future, TLS 1.2 and TLS 1.3 may be deprecated. To ensure that your application remains secure, avoid hardcoding a protocol version.
CA5399: Definitely disable HttpClient certificate revocation list check	A revoked certificate isn't trusted anymore. It could be used by attackers passing some malicious data or stealing sensitive data in HTTPS communication.
CA5400: Ensure HttpClient certificate revocation list check is not disabled	A revoked certificate isn't trusted anymore. It could be used by attackers passing some malicious data or stealing sensitive data in HTTPS communication.
CA5401: Do not use CreateEncryptor with non-default IV	Symmetric encryption should always use a non-repeatable initialization vector to prevent dictionary attacks.
CA5402: Use CreateEncryptor with the default IV	Symmetric encryption should always use a non-repeatable initialization vector to prevent dictionary attacks.

CA2100: Review SQL queries for security vulnerabilities

8/4/2020 • 5 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2100
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A method sets the System.Data.IDbCommand.CommandText property by using a string that is built from a string argument to the method.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

This rule assumes that any string, whose value can't be determined at compile time, may contain user input. A SQL command string that is built from user input is vulnerable to SQL injection attacks. In a SQL injection attack, a malicious user supplies input that alters the design of a query in an attempt to damage or gain unauthorized access to the underlying database. Typical techniques include injection of a single quotation mark or apostrophe, which is the SQL literal string delimiter; two dashes, which signifies a SQL comment; and a semicolon, which indicates that a new command follows. If user input must be part of the query, use one of the following, listed in order of effectiveness, to reduce the risk of attack.

- Use a stored procedure.
- Use a parameterized command string.
- Validate the user input for both type and content before you build the command string.

The following .NET types implement the CommandText property or provide constructors that set the property by using a string argument.

- System.Data.Odbc.OdbcCommand and System.Data.Odbc.OdbcDataAdapter
- System.Data.OleDb.OleDbCommand and System.Data.OleDb.OleDbDataAdapter
- System.Data.OracleClient.OracleCommand and System.Data.OracleClient.OracleDataAdapter
- System.Data.SqlClient.SqlCommand and System.Data.SqlClient.SqlDataAdapter

In some cases, this rule might not determine a string's value at compile time, even though you can. In those cases, this rule produces false positives when using those strings as SQL commands. The following is an example of such a string.

```
int x = 10;
string query = "SELECT TOP " + x.ToString() + " FROM Table";
```

The same applies when using `ToString()` implicitly.

```
int x = 10;
string query = String.Format("SELECT TOP {0} FROM Table", x);
```

How to fix violations

To fix a violation of this rule, use a parameterized query.

When to suppress warnings

It is safe to suppress a warning from this rule if the command text does not contain any user input.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2100.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2100.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2100.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2100.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2100.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Example

The following example shows a method, `UnsafeQuery`, that violates the rule and a method, `SaferQuery`, that satisfies the rule by using a parameterized command string.

```
Imports System
Imports System.Data
Imports System.Data.SqlClient

Namespace SecurityLibrary

   Public Class SqlQueries

      Function UnsafeQuery(connection As String, _
         name As String, password As String) As Object

         Dim someConnection As New SqlConnection(connection)
         Dim someCommand As New SqlCommand()
         someCommand.Connection = someConnection

         someCommand.CommandText = "SELECT AccountNumber FROM Users " & _
            "WHERE Username='" & name & "' AND Password='" & password & "'"

         someConnection.Open()
         Dim accountNumber As Object = someCommand.ExecuteScalar()
         someConnection.Close()
         Return accountNumber

      End Function

      Function SaferQuery(connection As String, _
         name As String, password As String) As Object

         Dim someConnection As New SqlConnection(connection)
         Dim someCommand As New SqlCommand()
         someCommand.Connection = someConnection

         someCommand.Parameters.Add( _
            "@username", SqlDbType.NChar).Value = name
         someCommand.Parameters.Add( _
            "@password", SqlDbType.NChar).Value = password
         someCommand.CommandText = "SELECT AccountNumber FROM Users " & _
            "WHERE Username=@username AND Password=@password"

         someConnection.Open()
         Dim accountNumber As Object = someCommand.ExecuteScalar()
         someConnection.Close()
         Return accountNumber

      End Function

   End Class

   Class MaliciousCode

      Shared Sub Main(args As String())

         Dim queries As New SqlQueries()
         queries.UnsafeQuery(args(0), "' OR 1=1 --", "anything")
         ' Resultant query (which is always true):
         ' SELECT AccountNumber FROM Users WHERE Username='' OR 1=1

         queries.SaferQuery(args(0), "' OR 1 = 1 --", "anything")
         ' Resultant query (notice the additional single quote character):
         ' SELECT AccountNumber FROM Users WHERE Username=''' OR 1=1 --'
         '                                     AND Password='anything'
      End Sub

   End Class

End Namespace
```

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace SecurityLibrary
{
   public class SqlQueries
   {
      public object UnsafeQuery(
         string connection, string name, string password)
      {
         SqlConnection someConnection = new SqlConnection(connection);
         SqlCommand someCommand = new SqlCommand();
         someCommand.Connection = someConnection;

         someCommand.CommandText = "SELECT AccountNumber FROM Users " +
            "WHERE Username='" + name +
            "' AND Password='" + password + "'";

         someConnection.Open();
         object accountNumber = someCommand.ExecuteScalar();
         someConnection.Close();
         return accountNumber;
      }

      public object SaferQuery(
         string connection, string name, string password)
      {
         SqlConnection someConnection = new SqlConnection(connection);
         SqlCommand someCommand = new SqlCommand();
         someCommand.Connection = someConnection;

         someCommand.Parameters.Add(
            "@username", SqlDbType.NChar).Value = name;
         someCommand.Parameters.Add(
            "@password", SqlDbType.NChar).Value = password;
         someCommand.CommandText = "SELECT AccountNumber FROM Users " +
            "WHERE Username=@username AND Password=@password";

         someConnection.Open();
         object accountNumber = someCommand.ExecuteScalar();
         someConnection.Close();
         return accountNumber;
      }
   }

   class MaliciousCode
   {
      static void Main(string[] args)
      {
         SqlQueries queries = new SqlQueries();
         queries.UnsafeQuery(args[0], "' OR 1=1 --", "anything");
         // Resultant query (which is always true):
         // SELECT AccountNumber FROM Users WHERE Username='' OR 1=1

         queries.SaferQuery(args[0], "' OR 1 = 1 --", "anything");
         // Resultant query (notice the additional single quote character):
         // SELECT AccountNumber FROM Users WHERE Username=''' OR 1=1 --'
         //                                     AND Password='anything'
      }
   }
}
```

```
#using <System.dll>
#using <System.Data.dll>
#using <System.EnterpriseServices.dll>
```

```
#using <System.Transactions.dll>
#using <System.Xml.dll>
using namespace System;
using namespace System::Data;
using namespace System::Data::SqlClient;

namespace SecurityLibrary
{
   public ref class SqlQueries
   {
   public:
      Object^ UnsafeQuery(
         String^ connection, String^ name, String^ password)
      {
         SqlConnection^ someConnection = gcnew SqlConnection(connection);
         SqlCommand^ someCommand = gcnew SqlCommand();
         someCommand->Connection = someConnection;

         someCommand->CommandText = String::Concat(
            "SELECT AccountNumber FROM Users WHERE Username='",
            name, "' AND Password='", password, "'");

         someConnection->Open();
         Object^ accountNumber = someCommand->ExecuteScalar();
         someConnection->Close();
         return accountNumber;
      }

      Object^ SaferQuery(
         String^ connection, String^ name, String^ password)
      {
         SqlConnection^ someConnection = gcnew SqlConnection(connection);
         SqlCommand^ someCommand = gcnew SqlCommand();
         someCommand->Connection = someConnection;

         someCommand->Parameters->Add(
            "@username", SqlDbType::NChar)->Value = name;
         someCommand->Parameters->Add(
            "@password", SqlDbType::NChar)->Value = password;
         someCommand->CommandText = "SELECT AccountNumber FROM Users "
            "WHERE Username=@username AND Password=@password";

         someConnection->Open();
         Object^ accountNumber = someCommand->ExecuteScalar();
         someConnection->Close();
         return accountNumber;
      }
   };
}

using namespace SecurityLibrary;

void main()
{
   SqlQueries^ queries = gcnew SqlQueries();
   queries->UnsafeQuery(Environment::GetCommandLineArgs()[1],
      "' OR 1=1 --", "anything");
   // Resultant query (which is always true):
   // SELECT AccountNumber FROM Users WHERE Username='' OR 1=1

   queries->SaferQuery(Environment::GetCommandLineArgs()[1],
      "' OR 1 = 1 --", "anything");
   // Resultant query (notice the additional single quote character):
   // SELECT AccountNumber FROM Users WHERE Username=''' OR 1=1 --'
   //                                    AND Password='anything'
}
```

See also

- Security Overview

CA2102: Catch non-CLSCompliant exceptions in general handlers

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2102
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A member in an assembly that is not marked with the RuntimeCompatibilityAttribute or is marked `RuntimeCompatibility(WrapNonExceptionThrows = false)` contains a catch block that handles System.Exception and does not contain an immediately following general catch block. This rule ignores Visual Basic assemblies.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A catch block that handles Exception catches all Common Language Specification (CLS) compliant exceptions. However, it does not catch non-CLS compliant exceptions. Non-CLS compliant exceptions can be thrown from native code or from managed code that was generated by the Microsoft intermediate language (MSIL) Assembler. Notice that the C# and Visual Basic compilers do not allow non-CLS compliant exceptions to be thrown and Visual Basic does not catch non-CLS compliant exceptions. If the intent of the catch block is to handle all exceptions, use the following general catch block syntax.

- C#: `catch {}`
- C++: `catch(...) {}` or `catch(Object^) {}`

An unhandled non-CLS compliant exception becomes a security issue when previously allowed permissions are removed in the catch block. Because non-CLS compliant exceptions are not caught, a malicious method that throws a non-CLS compliant exception could run with elevated permissions.

How to fix violations

To fix a violation of this rule when the intent is to catch all exceptions, substitute or add a general catch block or mark the assembly `RuntimeCompatibility(WrapNonExceptionThrows = true)`. If permissions are removed in the catch block, duplicate the functionality in the general catch block. If it is not the intent to handle all exceptions, replace the catch block that handles Exception with catch blocks that handle specific exception types.

When to suppress warnings

It is safe to suppress a warning from this rule if the try block does not contain any statements that might generate a non-CLS compliant exception. Because any native or managed code might throw a non-CLS compliant exception, this requires knowledge of all code that can be executed in all code paths inside the try block. Notice that non-CLS compliant exceptions are not thrown by the common language runtime.

Example 1

The following example shows an MSIL class that throws a non-CLS compliant exception.

```
.assembly ThrowNonClsCompliantException {}
.class public auto ansi beforefieldinit ThrowsExceptions
{
   .method public hidebysig static void
         ThrowNonClsException() cil managed
   {
      .maxstack  1
      IL_0000:  newobj     instance void [mscorlib]System.Object::.ctor()
      IL_0005:  throw
   }
}
```

Example 2

The following example shows a method that contains a general catch block that satisfies the rule.

```
// CatchNonClsCompliantException.cs
using System;

namespace SecurityLibrary
{
   class HandlesExceptions
   {
      void CatchAllExceptions()
      {
         try
         {
            ThrowsExceptions.ThrowNonClsException();
         }
         catch(Exception e)
         {
            // Remove some permission.
            Console.WriteLine("CLS compliant exception caught");
         }
         catch
         {
            // Remove the same permission as above.
            Console.WriteLine("Non-CLS compliant exception caught.");
         }
      }

      static void Main()
      {
         HandlesExceptions handleExceptions = new HandlesExceptions();
         handleExceptions.CatchAllExceptions();
      }
   }
}
```

Compile the previous examples as follows.

```
ilasm /dll ThrowNonClsCompliantException.il
csc /r:ThrowNonClsCompliantException.dll CatchNonClsCompliantException.cs
```

Related rules

CA1031: Do not catch general exception types

See also

- Exceptions and Exception Handling
- Ilasm.exe (IL Assembler)
- Language Independence and Language-Independent Components

CA2103: Review imperative security

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2103
Category	Microsoft.Security
Breaking change	Breaking

Cause

A method uses imperative security and might be constructing the permission by using state information or return values that can change as long as the demand is active.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Imperative security uses managed objects to specify permissions and security actions during code execution, compared to declarative security, which uses attributes to store permissions and actions in metadata. Imperative security is flexible because you can set the state of a permission object and select security actions by using information that is not available until run time. Together with that flexibility comes the risk that the run-time information that you use to determine the state of a permission does not remain unchanged as long as the action is in effect.

Use declarative security whenever possible. Declarative demands are easier to understand.

How to fix violations

Review the imperative security demands to make sure that the state of the permission does not rely on information that can change as long as the permission is being used.

When to suppress warnings

It is safe to suppress a warning from this rule if the permission does not rely on changing data. However, it is better to change the imperative demand to its declarative equivalent.

See also

- Secure Coding Guidelines
- Data and Modeling

CA2104: Do not declare read only mutable reference types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2104
Category	Microsoft.Security
Breaking change	Non-breaking

NOTE

Rule CA2104 is obsolete and will be removed in a future version of Visual Studio. It will not be implemented as an analyzer due to the complicated analysis that's required to determine the actual immutability of a type.

Cause

An externally visible type contains an externally visible read-only field that is a mutable reference type.

NOTE

This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A mutable type is a type whose instance data can be modified. The System.Text.StringBuilder class is an example of a mutable reference type. It contains members that can change the value of an instance of the class. An example of an immutable reference type is the System.String class. After it has been instantiated, its value can never change.

The read-only modifier (readonly in C#, ReadOnly in Visual Basic, and const in C++) on a reference type field (or pointer in C++) prevents the field from being replaced by a different instance of the reference type. However, the modifier does not prevent the instance data of the field from being modified through the reference type.

This rule may inadvertently show a violation for a type that is, in fact, immutable. In that case, it's safe to suppress the warning.

Read-only array fields are exempt from this rule but instead cause a violation of the CA2105: Array fields should not be read only rule.

How to fix violations

To fix a violation of this rule, remove the read-only modifier or, if a breaking change is acceptable, replace the field with an immutable type.

When to suppress warnings

It's safe to suppress a warning from this rule if the field type is immutable.

Example

The following example shows a field declaration that causes a violation of this rule:

```
using namespace System;
using namespace System::Text;

namespace SecurityLibrary
{
    public ref class MutableReferenceTypes
    {
    protected:
        static StringBuilder^ const SomeStringBuilder =
           gcnew StringBuilder();

    private:
        static MutableReferenceTypes()
        {
        }
    };
}
```

```
using System;
using System.Text;

namespace SecurityLibrary
{
    public class MutableReferenceTypes
    {
        static protected readonly StringBuilder SomeStringBuilder;

        static MutableReferenceTypes()
        {
            SomeStringBuilder = new StringBuilder();
        }
    }
}
```

```
Imports System
Imports System.Text

Namespace SecurityLibrary

    Public Class MutableReferenceTypes

        Shared Protected ReadOnly SomeStringBuilder As StringBuilder

        Shared Sub New()
            SomeStringBuilder = New StringBuilder()
        End Sub

    End Class

End Namespace
```

CA2105: Array fields should not be read only

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2105
Category	Microsoft.Security
Breaking change	Breaking

Cause

A public or protected field that holds an array is declared read-only.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

When you apply the `readonly` (`ReadOnly` in Visual Basic) modifier to a field that contains an array, the field cannot be changed to refer to a different array. However, the elements of the array that are stored in a read-only field can be changed. Code that makes decisions or performs operations that are based on the elements of a read-only array that can be publicly accessed might contain an exploitable security vulnerability.

Note that having a public field also violates the design rule CA1051: Do not declare visible instance fields.

How to fix violations

To fix the security vulnerability that is identified by this rule, do not rely on the contents of a read-only array that can be publicly accessed. It is strongly recommended that you use one of the following procedures:

- Replace the array with a strongly typed collection that cannot be changed. For more information, see System.Collections.ReadOnlyCollectionBase.
- Replace the public field with a method that returns a clone of a private array. Because your code does not rely on the clone, there is no danger if the elements are modified.

If you chose the second approach, do not replace the field with a property; properties that return arrays adversely affect performance. For more information, see CA1819: Properties should not return arrays.

When to suppress warnings

Exclusion of a warning from this rule is strongly discouraged. Almost no scenarios occur where the contents of a read-only field are unimportant. If this is the case with your scenario, remove the `readonly` modifier instead of excluding the message.

Example 1

This example demonstrates the dangers of violating this rule. The first part shows an example library that has a type, `MyClassWithReadOnlyArrayField`, that contains two fields (`grades` and `privateGrades`) that are not secure. The field `grades` is public, and therefore vulnerable to any caller. The field `privateGrades` is private but is still vulnerable because it is returned to callers by the `GetPrivateGrades` method. The `securePrivateGrades` field is exposed in a safe manner by the `GetSecurePrivateGrades` method. It is declared as private to follow good design practices. The second part shows code that changes values stored in the `grades` and `privateGrades` members.

The example class library appears in the following example.

```
using System;

namespace SecurityRulesLibrary
{
   public class MyClassWithReadOnlyArrayField
   {
      public readonly int[] grades = {90, 90, 90};
      private readonly int[] privateGrades = {90, 90, 90};
      private readonly int[] securePrivateGrades = {90, 90, 90};

      // Making the array private does not protect it because it is passed to others.
      public int[] GetPrivateGrades()
      {
         return privateGrades;
      }
      //This method secures the array by cloning it.
      public int[] GetSecurePrivateGrades()
      {
            return (int[])securePrivateGrades.Clone();
      }

      public override string ToString()
      {
         return String.Format("Grades: {0}, {1}, {2} Private Grades: {3}, {4}, {5}  Secure Grades, {6}, {7},
{8}",
            grades[0], grades[1], grades[2], privateGrades[0], privateGrades[1], privateGrades[2],
securePrivateGrades[0], securePrivateGrades[1], securePrivateGrades[2]);
      }
   }
}
```

Example 2

The following code uses the example class library to illustrate read-only array security issues.

```
using System;
using SecurityRulesLibrary;

namespace TestSecRulesLibrary
{
   public class TestArrayReadOnlyRule
   {
      [STAThread]
      public static void Main()
      {
         MyClassWithReadOnlyArrayField dataHolder =
            new MyClassWithReadOnlyArrayField();

         // Get references to the library's readonly arrays.
         int[] theGrades = dataHolder.grades;
         int[] thePrivateGrades = dataHolder.GetPrivateGrades();
         int[] theSecureGrades = dataHolder.GetSecurePrivateGrades();

         Console.WriteLine(
            "Before tampering: {0}", dataHolder.ToString());

         // Overwrite the contents of the "readonly" array.
         theGrades[1]= 555;
         thePrivateGrades[1]= 555;
         theSecureGrades[1]= 555;
         Console.WriteLine(
            "After tampering: {0}",dataHolder.ToString());
      }
   }
}
```

The output from this example is:

```
Before tampering: Grades: 90, 90, 90 Private Grades: 90, 90, 90  Secure Grades, 90, 90, 90
After tampering: Grades: 90, 555, 90 Private Grades: 90, 555, 90  Secure Grades, 90, 90, 90
```

Related rules

- CA2104 : Do not declare read only mutable reference types

See also

- System.Array
- System.Collections.ReadOnlyCollectionBase

CA2106: Secure asserts

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2106
Category	Microsoft.Security
Breaking change	Breaking

Cause

A method asserts a permission and performs no security checks on the caller.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Asserting a security permission without performing any security checks can leave an exploitable security weakness in your code. A security stack walk stops when a security permission is asserted. If you assert a permission without performing any checks on the caller, the caller could indirectly execute code by using your permissions. Asserts without security checks are permissible if you're sure the assert can't be used in a harmful manner. An assert is harmless if the code you call is harmless, or if users can't pass arbitrary information to code that you call.

How to fix violations

To fix a violation of this rule, add a security demand to the method or its declaring type.

When to suppress warnings

Suppress a warning from this rule only after a careful security review.

See also

- System.Security.CodeAccessPermission.Assert
- Secure Coding Guidelines

CA2107: Review deny and permit only usage

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2107
Category	Microsoft.Security
Breaking change	Breaking

Cause

A method contains a security check that specifies the PermitOnly or Deny security action.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

The System.Security.CodeAccessPermission.Deny security action should be used only by those who have an advanced knowledge of .NET security. Code that uses these security actions should undergo a security review.

Deny alters the default behavior of the stack walk that occurs in response to a security demand. It lets you specify permissions that must not be granted for the duration of the denying method, regardless of the actual permissions of the callers in the call stack. If the stack walk detects a method that is secured by Deny, and if the demanded permission is included in the denied permissions, the stack walk fails. PermitOnly also alters the default behavior of the stack walk. It allows code to specify only those permissions that can be granted, regardless of the permissions of the callers. If the stack walk detects a method that is secured by PermitOnly, and if the demanded permission is not included in the permissions that are specified by the PermitOnly, the stack walk fails.

Code that relies on these actions should be carefully evaluated for security vulnerabilities because of their limited usefulness and subtle behavior. Consider the following:

- Link Demands are not affected by Deny or PermitOnly.
- If the Deny or PermitOnly occurs in the same stack frame as the demand that causes the stack walk, the security actions have no effect.
- Values that are used to construct path-based permissions can usually be specified in multiple ways. Denying access to one form of the path does not deny access to all forms. For example, if a file share \\Server\Share is mapped to a network drive X:, to deny access to a file on the share, you must deny \\Server\Share\File, X:\File, and every other path that accesses the file.
- An System.Security.CodeAccessPermission.Assert can terminate a stack walk before the Deny or PermitOnly is reached.
- If a Deny has any effect, namely, when a caller has a permission that is blocked by the Deny, the caller can access the protected resource directly, bypassing the Deny. Similarly, if the caller does not have the denied permission, the stack walk would fail without the Deny.

How to fix violations

Any use of these security actions will cause a violation. To fix a violation, do not use these security actions.

When to suppress warnings

Suppress a warning from this rule only after you complete a security review.

Example 1

The following example demonstrates some limitations of Deny. The library contains a class that has two methods that are identical except for the security demands that protect them.

```
using System.Security;
using System.Security.Permissions;
using System;

namespace SecurityRulesLibrary
{
   public  class SomeSecuredMethods
   {

      // Demand immediate caller has suitable permission
      // before revealing sensitive data.
      [EnvironmentPermissionAttribute(SecurityAction.LinkDemand,
          Read="COMPUTERNAME;USERNAME;USERDOMAIN")]

      public static void MethodProtectedByLinkDemand()
      {
         Console.Write("LinkDemand: ");
      }

      [EnvironmentPermissionAttribute(SecurityAction.Demand,
          Read="COMPUTERNAME;USERNAME;USERDOMAIN")]

      public static void MethodProtectedByDemand()
      {
         Console.Write("Demand: ");
      }
   }
}
```

Example 2

The following application demonstrates the effects of Deny on the secured methods from the library.

```
using System.Security;
using System.Security.Permissions;
using System;
using SecurityRulesLibrary;

namespace TestSecurityLibrary
{
    // Violates rule: ReviewDenyAndPermitOnlyUsage.
   public class TestPermitAndDeny
   {
      public static void TestAssertAndDeny()
      {
         EnvironmentPermission envPermission = new EnvironmentPermission(
               EnvironmentPermissionAccess.Read,
               "COMPUTERNAME;USERNAME;USERDOMAIN");
         envPermission.Assert();
```

```
         try
         {
            SomeSecuredMethods.MethodProtectedByDemand();
            Console.WriteLine(
               "Caller's Deny has no effect on Demand " +
               "with the asserted permission.");

            SomeSecuredMethods.MethodProtectedByLinkDemand();
            Console.WriteLine(
               "Caller's Deny has no effect on LinkDemand " +
               "with the asserted permission.");
         }
         catch (SecurityException e)
         {
            Console.WriteLine(
               "Caller's Deny protected the library.{0}", e);
         }
      }

      public static void TestDenyAndLinkDemand()
      {
         try
         {
            SomeSecuredMethods.MethodProtectedByLinkDemand();
            Console.WriteLine(
               "Caller's Deny has no effect with " +
               "LinkDemand-protected code.");
         }
         catch (SecurityException e)
         {
            Console.WriteLine(
               "Caller's Deny protected the library.{0}",e);
         }
      }

      public static void Main()
      {
         EnvironmentPermission envPermission = new EnvironmentPermission(
            EnvironmentPermissionAccess.Read,
            "COMPUTERNAME;USERNAME;USERDOMAIN");
         envPermission.Deny();

         //Test Deny and Assert interaction for LinkDemands and Demands.
         TestAssertAndDeny();

         //Test Deny's effects on code in different stack frame.
         TestDenyAndLinkDemand();

         //Test Deny's effect on code in same frame as deny.
         try
         {
            SomeSecuredMethods.MethodProtectedByLinkDemand();
            Console.WriteLine(
               "This Deny has no effect with LinkDemand-protected code.");
         }
         catch (SecurityException e)
         {
            Console.WriteLine("This Deny protected the library.{0}",e);
         }
      }
   }
}
```

This example produces the following output:

```
Demand: Caller's Deny has no effect on Demand with the asserted permission.
LinkDemand: Caller's Deny has no effect on LinkDemand with the asserted permission.
LinkDemand: Caller's Deny has no effect with LinkDemand-protected code.
LinkDemand: This Deny has no effect with LinkDemand-protected code.
```

See also

- System.Security.CodeAccessPermission.PermitOnly
- System.Security.CodeAccessPermission.Assert
- System.Security.CodeAccessPermission.Deny
- System.Security.IStackWalk.PermitOnly
- Secure Coding Guidelines

CA2108: Review declarative security on value types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2108
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A public or protected value type is secured by a Data and Modeling or Link Demands.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Value types are allocated and initialized by their default constructors before other constructors execute. If a value type is secured by a Demand or LinkDemand, and the caller does not have permissions that satisfy the security check, any constructor other than the default will fail, and a security exception will be thrown. The value type is not deallocated; it is left in the state set by its default constructor. Do not assume that a caller that passes an instance of the value type has permission to create or access the instance.

How to fix violations

You cannot fix a violation of this rule unless you remove the security check from the type, and use method level security checks in its place. Fixing the violation in this manner does not prevent callers with inadequate permissions from obtaining instances of the value type. You must ensure that an instance of the value type, in its default state, does not expose sensitive information, and cannot be used in a harmful manner.

When to suppress warnings

You can suppress a warning from this rule if any caller can obtain instances of the value type in its default state without posing a threat to security.

Example 1

The following example shows a library containing a value type that violates this rule. The `StructureManager` type assumes that a caller that passes an instance of the value type has permission to create or access the instance.

```
using System;
using System.Security;
using System.Security.Permissions;

[assembly:AllowPartiallyTrustedCallers]

namespace SecurityRulesLibrary
{
   // Violates rule: ReviewDeclarativeSecurityOnValueTypes.
   [System.Security.Permissions.PermissionSetAttribute(System.Security.Permissions.SecurityAction.Demand,
Name="FullTrust")]

   public struct SecuredTypeStructure
   {
      internal double xValue;
      internal double yValue;

      public SecuredTypeStructure(double x, double y)
      {
         xValue = x;
         yValue = y;
         Console.WriteLine("Creating an instance of SecuredTypeStructure.");
      }
      public override string ToString()
      {
         return String.Format ("SecuredTypeStructure {0} {1}", xValue, yValue);
      }
   }

   public class StructureManager
   {
      // This method asserts trust, incorrectly assuming that the caller must have
      // permission because they passed in instance of the value type.
      [System.Security.Permissions.PermissionSetAttribute(System.Security.Permissions.SecurityAction.Assert,
Name="FullTrust")]

      public static SecuredTypeStructure AddStepValue(SecuredTypeStructure aStructure)
      {
         aStructure.xValue += 100;
         aStructure.yValue += 100;
         Console.WriteLine ("New values {0}", aStructure.ToString());
         return aStructure;
      }
   }
}
```

Example 2

The following application demonstrates the library's weakness.

```
using System;
using System.Runtime.InteropServices;
using System.Security;
using System.Security.Permissions;
using SecurityRulesLibrary;

// Run this test code with partial trust.
[assembly: System.Security.Permissions.PermissionSetAttribute(
   System.Security.Permissions.SecurityAction.RequestRefuse,
   Name="FullTrust")]

namespace TestSecurityExamples
{
   public class TestDemandOnValueType
   {
      static SecuredTypeStructure mystruct;

      [STAThread]
      public static void Main()
      {
         try
         {
            mystruct = new SecuredTypeStructure(10,10);

         }
         catch (SecurityException e)
         {
            Console.WriteLine(
               "Structure custom constructor: {0}", e.Message);
         }

         // The call to the default constructor
         // does not throw an exception.
         try
         {
            mystruct = StructureManager.AddStepValue(
               new SecuredTypeStructure());
         }
         catch (SecurityException e)
         {
            Console.WriteLine(
               "Structure default constructor: {0}", e.Message);
         }

         try
         {
            StructureManager.AddStepValue(mystruct);
         }

         catch (Exception e)
         {
            Console.WriteLine(
               "StructureManager add step: {0}", e.Message);
         }
      }
   }
}
```

This example produces the following output:

```
Structure custom constructor: Request failed.
New values SecuredTypeStructure 100 100
New values SecuredTypeStructure 200 200
```

See also

- Link Demands
- Data and Modeling

CA2109: Review visible event handlers

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2109
Category	Microsoft.Security
Breaking change	Breaking

Cause

A public or protected event-handling method was detected.

Rule description

An externally visible event-handling method presents a security issue that requires review.

Do not expose event-handling methods unless absolutely necessary. An event handler, a delegate type, that invokes the exposed method can be added to any event as long as the handler and event signatures match. Events can potentially be raised by any code, and are frequently raised by highly trusted system code in response to user actions such as clicking a button. Adding a security check to an event-handling method does not prevent code from registering an event handler that invokes the method.

A demand cannot reliably protect a method invoked by an event handler. Security demands help protect code from untrusted callers by examining the callers on the call stack. Code that adds an event handler to an event is not necessarily present on the call stack when the event handler's methods run. Therefore, the call stack might have only highly trusted callers when the event handler method is invoked. This causes demands made by the event handler method to succeed. Also, the demanded permission might be asserted when the method is invoked. For these reasons, the risk of not fixing a violation of this rule can only be assessed after reviewing the event-handling method. When you review your code, consider the following issues:

- Does your event handler perform any operations that are dangerous or exploitable, such as asserting permissions or suppressing unmanaged code permission?
- What are the security threats to and from your code because it can run at any time with only highly trusted callers on the stack?

How to fix violations

To fix a violation of this rule, review the method and evaluate the following:

- Can you make the event-handling method non-public?
- Can you move all dangerous functionality out of the event handler?
- If a security demand is imposed, can this be accomplished in some other manner?

When to suppress warnings

Suppress a warning from this rule only after a careful security review to make sure that your code does not pose a

security threat.

Example

The following code shows an event-handling method that can be misused by malicious code.

```
using System;
using System.Security;
using System.Security.Permissions;

namespace EventSecLibrary
{
   public class HandleEvents
   {
      // Due to the access level and signature, a malicious caller could
      // add this method to system-triggered events where all code in the call
      // stack has the demanded permission.

      // Also, the demand might be canceled by an asserted permission.

      [SecurityPermissionAttribute(SecurityAction.Demand, UnmanagedCode=true)]

      // Violates rule: ReviewVisibleEventHandlers.
      public static void SomeActionHappened(Object sender, EventArgs e)
      {
         Console.WriteLine ("Do something dangerous from unmanaged code.");
      }

   }
}
```

See also

- System.Security.CodeAccessPermission.Demand
- System.EventArgs

CA2111: Pointers should not be visible

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2111
Category	Microsoft.Security
Breaking change	Breaking

Cause

A public or protected System.IntPtr or System.UIntPtr field is not read-only.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

IntPtr and UIntPtr are pointer types that are used to access unmanaged memory. If a pointer is not private, internal, or read-only, malicious code can change the value of the pointer, potentially allowing access to arbitrary locations in memory or causing application or system failures.

If you intend to secure access to the type that contains the pointer field, see CA2112: Secured types should not expose fields.

How to fix violations

Secure the pointer by making it read-only, internal, or private.

When to suppress warnings

Suppress a warning from this rule if you do not rely on the value of the pointer.

Example

The following code shows pointers that violate and satisfy the rule. Notice that the non-private pointers also violate the rule CA1051: Do not declare visible instance fields.

```
using System;

namespace SecurityRulesLibrary
{

   public class ExposedPointers
   {
      // Violates rule: PointersShouldNotBeVisible.
      public IntPtr publicPointer1;
      public UIntPtr publicPointer2;
      protected IntPtr protectedPointer;

      // Satisfies the rule.
      internal UIntPtr internalPointer;
      private UIntPtr privatePointer;

      public readonly UIntPtr publicReadOnlyPointer;
      protected readonly IntPtr protectedReadOnlyPointer;
   }
}
```

Related rules

CA2112: Secured types should not expose fields

CA1051: Do not declare visible instance fields

See also

- System.IntPtr
- System.UIntPtr

CA2112: Secured types should not expose fields

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2112
Category	Microsoft.Security
Breaking change	Breaking

Cause

A public or protected type contains public fields and is secured by a Link Demands.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

If code has access to an instance of a type that is secured by a link demand, the code does not have to satisfy the link demand to access the type's fields.

How to fix violations

To fix a violation of this rule, make the fields nonpublic and add public properties or methods that return the field data. LinkDemand security checks on types protect access to the type's properties and methods. However, code access security does not apply to fields.

When to suppress warnings

Both for security issues and for good design, you should fix violations by making the public fields nonpublic. You can suppress a warning from this rule if the field does not hold information that should remain secured, and you do not rely on the contents of the field.

Example

The following example is composed of a library type (`SecuredTypeWithFields`) with unsecured fields, a type (`Distributor`) that can create instances of the library type and mistaken passes instances to types do not have permission to create them, and application code that can read an instance's fields even though it does not have the permission that secures the type.

The following library code violates the rule.

```
using System;
using System.Reflection;
using System.Security;
using System.Security.Permissions;

namespace SecurityRulesLibrary
{
   // This code requires immediate callers to have full trust.
   [System.Security.Permissions.PermissionSetAttribute(
       System.Security.Permissions.SecurityAction.LinkDemand,
       Name="FullTrust")]
   public class SecuredTypeWithFields
   {
      // Even though the type is secured, these fields are not.
      // Violates rule: SecuredTypesShouldNotExposeFields.
      public double xValue;
      public double yValue;

      public SecuredTypeWithFields (double x, double y)
      {
         xValue = x;
         yValue = y;
         Console.WriteLine(
            "Creating an instance of SecuredTypeWithFields.");
      }
      public override string ToString()
      {
          return String.Format (
            "SecuredTypeWithFields {0} {1}", xValue, yValue);
      }
   }
}
```

Example 1

The application cannot create an instance because of the link demand that protects the secured type. The following class enables the application to obtain an instance of the secured type.

```
using System;
using System.Reflection;
using System.Security;
using System.Security.Permissions;

// This assembly executes with full trust.

namespace SecurityRulesLibrary
{
   // This type creates and returns instances of the secured type.
   // The GetAnInstance method incorrectly gives the instance
   // to a type that does not have the link demanded permission.

   public class Distributor
   {
      static SecuredTypeWithFields s = new SecuredTypeWithFields(22,33);
      public static SecuredTypeWithFields GetAnInstance ()
      {
            return s;
      }

      public static void DisplayCachedObject ()
      {
         Console.WriteLine(
            "Cached Object fields: {0}, {1}", s.xValue , s.yValue);
      }
   }
}
```

Example 2

The following application illustrates how, without permission to access a secured type's methods, code can access its fields.

```
using System;
using System.Security;
using System.Security.Permissions;
using SecurityRulesLibrary;

// This code executes with partial trust.
[assembly: System.Security.Permissions.PermissionSetAttribute(
   System.Security.Permissions.SecurityAction.RequestRefuse,
   Name = "FullTrust")]
namespace TestSecurityExamples
{
    public class TestLinkDemandOnField
    {
        [STAThread]
        public static void Main()
        {
            // Get an instance of the protected object.
            SecuredTypeWithFields secureType = Distributor.GetAnInstance();

            // Even though this type does not have full trust,
            // it can directly access the secured type's fields.
            Console.WriteLine(
               "Secured type fields: {0}, {1}",
               secureType.xValue,
               secureType.yValue);
            Console.WriteLine("Changing secured type's field...");
            secureType.xValue = 99;

            // Distributor must call ToString on the secured object.
            Distributor.DisplayCachedObject();

            // If the following line is uncommented, a security
            // exception is thrown at JIT-compilation time because
            // of the link demand for full trust that protects
            // SecuredTypeWithFields.ToString().

            // Console.WriteLine("Secured type {0}",secureType.ToString());
        }
    }
}
```

This example produces the following output:

```
Creating an instance of SecuredTypeWithFields.
Secured type fields: 22, 33
Changing secured type's field...
Cached Object fields: 99, 33
```

Related rules

- CA1051: Do not declare visible instance fields

See also

- Link Demands
- Data and Modeling

CA2114: Method security should be a superset of type

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2114
Category	Microsoft.Security
Breaking change	Breaking

Cause

A type has declarative security and one of its methods has declarative security for the same security action, and the security action is not Link Demands, and the permissions checked by the type are not a subset of the permissions checked by the method.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A method should not have both a method-level and type-level declarative security for the same action. The two checks are not combined; only the method-level demand is applied. For example, if a type demands permission `X`, and one of its methods demands permission `Y`, code does not have to have permission `X` to execute the method.

How to fix violations

Review your code to make sure that both actions are required. If both actions are required, make sure that the method-level action includes the security specified at the type level. For example, if your type demands permission `X`, and its method must also demand permission `Y`, the method should explicitly demand `X` and `Y`.

When to suppress warnings

It is safe to suppress a warning from this rule if the method does not require the security specified by the type. However, this is not an ordinary scenario and might indicate a need for a careful design review.

Example 1

The following example uses environment permissions to demonstrate the dangers of violating this rule. In this example, the application code creates an instance of the secured type before denying the permission required by the type. In a real-world threat scenario, the application would require another way to obtain an instance of the object.

In the following example, the library demands write permission for a type and read permission for a method.

```
using System;
using System.Security;
using System.Security.Permissions;
using System.Runtime.InteropServices;

namespace SecurityRulesLibrary
{
   [EnvironmentPermissionAttribute(SecurityAction.Demand, Write="PersonalInfo")]
   public class MyClassWithTypeSecurity
   {
      [DllImport("kernel32.dll", CharSet=CharSet.Unicode, SetLastError=true)]
      [return:MarshalAs(UnmanagedType.Bool)]
      public static extern bool SetEnvironmentVariable(
         string lpName,
         string lpValue);

      // Constructor.
      public MyClassWithTypeSecurity(int year, int month, int day)
      {
         DateTime birthday = new DateTime(year, month, day);

         // Write out PersonalInfo environment variable.
         SetEnvironmentVariable("PersonalInfo",birthday.ToString());
      }

      [EnvironmentPermissionAttribute(SecurityAction.Demand, Read="PersonalInfo")]
      public string PersonalInformation ()
      {
         // Read the variable.
         return Environment.GetEnvironmentVariable("PersonalInfo");
      }
   }
}
```

Example 2

The following application code demonstrates the vulnerability of the library by calling the method even though it does not meet the type-level security requirement.

```
using System;
using System.Security;
using System.Security.Permissions;
using SecurityRulesLibrary;

namespace TestSecRulesLibrary
{
   public class TestMethodLevelSecurity
   {
      MyClassWithTypeSecurity dataHolder;

      void RetrievePersonalInformation(string description)
      {
         try
         {
            Console.WriteLine(
               "{0} Personal information: {1}",
               description, dataHolder.PersonalInformation());
         }
         catch (SecurityException e)
         {
            Console.WriteLine(
               "{0} Could not access personal information: {1}",
               description, e.Message);
         }
      }

      [STAThread]
      public static void Main()
      {
         TestMethodLevelSecurity me = new TestMethodLevelSecurity();

         me.dataHolder = new MyClassWithTypeSecurity(1964,06,16);

         // Local computer zone starts with all environment permissions.
         me.RetrievePersonalInformation("[All permissions]");

         // Deny the write permission required by the type.
         EnvironmentPermission epw = new EnvironmentPermission(
            EnvironmentPermissionAccess.Write,"PersonalInfo");
         epw.Deny();

         // Even though the type requires write permission,
         // and you do not have it; you can get the data.
         me.RetrievePersonalInformation(
            "[No write permission (demanded by type)]");

         // Reset the permissions and try to get
         // data without read permission.
         CodeAccessPermission.RevertAll();

         // Deny the read permission required by the method.
         EnvironmentPermission epr = new EnvironmentPermission(
            EnvironmentPermissionAccess.Read,"PersonalInfo");
         epr.Deny();

         // The method requires read permission, and you
         // do not have it; you cannot get the data.
         me.RetrievePersonalInformation(
            "[No read permission (demanded by method)]");
      }
   }
}
```

This example produces the following output:

```
[All permissions] Personal information: 6/16/1964 12:00:00 AM
[No write permission (demanded by type)] Personal information: 6/16/1964 12:00:00 AM
[No read permission (demanded by method)] Could not access personal information: Request failed.
```

See also

- Secure Coding Guidelines
- Link Demands
- Data and Modeling

CA2115: Call GC.KeepAlive when using native resources

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2115
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A method declared in a type with a finalizer references a System.IntPtr or System.UIntPtr field, but does not call System.GC.KeepAlive.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Garbage collection finalizes an object if there are no more references to it in managed code. Unmanaged references to objects do not prevent garbage collection. This rule detects errors that might occur because an unmanaged resource is being finalized while it is still being used in unmanaged code.

This rule assumes that IntPtr and UIntPtr fields store pointers to unmanaged resources. Because the purpose of a finalizer is to free unmanaged resources, the rule assumes that the finalizer will free the unmanaged resource pointed to by the pointer fields. This rule also assumes that the method is referencing the pointer field to pass the unmanaged resource to unmanaged code.

How to fix violations

To fix a violation of this rule, add a call to KeepAlive to the method, passing the current instance (`this` in C# and C++) as the argument. Position the call after the last line of code where the object must be protected from garbage collection. Immediately after the call to KeepAlive, the object is again considered ready for garbage collection assuming that there are no managed references to it.

When to suppress warnings

This rule makes some assumptions that can lead to false positives. You can safely suppress a warning from this rule if:

- The finalizer does not free the contents of the IntPtr or UIntPtr field referenced by the method.
- The method does not pass the IntPtr or UIntPtr field to unmanaged code.

Carefully review other messages before excluding them. This rule detects errors that are difficult to reproduce

and debug.

Example

In the following example, `BadMethod` does not include a call to `GC.KeepAlive` and therefore violates the rule. `GoodMethod` contains the corrected code.

> **NOTE**
>
> This example is pseudo-code. Although the code compiles and runs, the warning is not fired because an unmanaged resource is not created or freed.

```
using System;

namespace SecurityRulesLibrary
{
   class IntPtrFieldsAndFinalizeRequireGCKeepAlive
   {
      private IntPtr unmanagedResource;

      IntPtrFieldsAndFinalizeRequireGCKeepAlive()
      {
         GetUnmanagedResource (unmanagedResource);
      }

      // The finalizer frees the unmanaged resource.
      ~IntPtrFieldsAndFinalizeRequireGCKeepAlive()
      {
         FreeUnmanagedResource (unmanagedResource);
      }

      // Violates rule:CallGCKeepAliveWhenUsingNativeResources.
      void BadMethod()
      {
         // Call some unmanaged code.
         CallUnmanagedCode(unmanagedResource);
      }

      // Satisfies the rule.
      void GoodMethod()
      {
         // Call some unmanaged code.
         CallUnmanagedCode(unmanagedResource);
         GC.KeepAlive(this);
      }

      // Methods that would typically make calls to unmanaged code.
      void GetUnmanagedResource(IntPtr p)
      {
        // Allocate the resource ...
      }
      void FreeUnmanagedResource(IntPtr p)
      {
        // Free the resource and set the pointer to null ...
      }
      void CallUnmanagedCode(IntPtr p)
      {
        // Use the resource in unmanaged code ...
      }

   }

}
```

See also

- System.GC.KeepAlive
- System.IntPtr
- System.Object.Finalize
- System.UIntPtr
- Dispose Pattern

CA2116: APTCA methods should only call APTCA methods

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2116
Category	Microsoft.Security
Breaking change	Breaking

Cause

A method in an assembly with the System.Security.AllowPartiallyTrustedCallersAttribute attribute calls a method in an assembly that does not have the attribute.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

By default, public or protected methods in assemblies with strong names are implicitly protected by a Link Demands for full trust; only fully trusted callers can access a strong-named assembly. Strong-named assemblies marked with the AllowPartiallyTrustedCallersAttribute (APTCA) attribute do not have this protection. The attribute disables the link demand, making the assembly accessible to callers that do not have full trust, such as code executing from an intranet or the Internet.

When the APTCA attribute is present on a fully trusted assembly, and the assembly executes code in another assembly that does not allow partially trusted callers, a security exploit is possible. If two methods `M1` and `M2` meet the following conditions, malicious callers can use the method `M1` to bypass the implicit full trust link demand that protects `M2`:

- `M1` is a public method declared in a fully trusted assembly that has the APTCA attribute.
- `M1` calls a method `M2` outside `M1`'s assembly.
- `M2`'s assembly does not have the APTCA attribute and, therefore, should not be executed by or on behalf of callers that are partially trusted.

A partially trusted caller `X` can call method `M1`, causing `M1` to call `M2`. Because `M2` does not have the APTCA attribute, its immediate caller (`M1`) must satisfy a link demand for full trust; `M1` has full trust and therefore satisfies this check. The security risk is because `X` does not participate in satisfying the link demand that protects `M2` from untrusted callers. Therefore, methods with the APTCA attribute must not call methods that do not have the attribute.

How to fix violations

If the APCTA attribute is required, use a demand to protect the method that calls into the full trust assembly. The exact permissions you demand will depend on the functionality exposed by your method. If it is possible, protect the method with a demand for full trust to ensure that the underlying functionality is not exposed to partially trusted callers. If this is not possible, select a set of permissions that effectively protects the exposed functionality.

When to suppress warnings

To safely suppress a warning from this rule, you must ensure that the functionality exposed by your method does not directly or indirectly allow callers to access sensitive information, operations, or resources that can be used in a destructive manner.

Example 1

The following example uses two assemblies and a test application to illustrate the security vulnerability detected by this rule. The first assembly does not have the APTCA attribute and should not be accessible to partially trusted callers (represented by `M2` in the previous discussion).

```
using System;
using System.Security;
using System.Security.Permissions;
using System.Reflection;

// This code is compiled into a strong-named
// assembly that requires full trust and does
// not allow partially trusted callers.

namespace AptcaTestLibrary
{
   public class ClassRequiringFullTrust
   {
      public static void DoWork()
      {
        Console.WriteLine("ClassRequiringFullTrust.DoWork was called.");
      }
   }
}
```

Example 2

The second assembly is fully trusted and allows partially trusted callers (represented by `M1` in the previous discussion).

```
using System;
using System.Security;
using System.Security.Permissions;
using System.Reflection;

// This assembly executes with full trust and
// allows partially trusted callers.

[assembly:AllowPartiallyTrustedCallers]

namespace AptcaTestLibrary
{
   public class AccessAClassRequiringFullTrust
   {
      public static void Access()
      {
         // This security check fails if the caller
         // does not have full trust.
         NamedPermissionSet pset= new NamedPermissionSet("FullTrust");

         // This try-catch block shows the caller's permissions.
         // Correct code would either not catch the exception,
         // or would rethrow it.
         try
         {
            pset.Demand();
         }
         catch (SecurityException e)
         {
            Console.WriteLine("Demand for full trust:{0}", e.Message);
         }
         // Call the type that requires full trust.
         // Violates rule AptcaMethodsShouldOnlyCallAptcaMethods.
         ClassRequiringFullTrust.DoWork();
     }
   }
}
```

Example 3

The test application (represented by `x` in the previous discussion) is partially trusted.

```
using System;
using AptcaTestLibrary;

// If this test is run from the local computer, it gets full trust by default.
// Remove full trust.
[assembly:System.Security.Permissions.PermissionSetAttribute(
   System.Security.Permissions.SecurityAction.RequestRefuse, Name="FullTrust")]

namespace TestSecLibrary
{
   class TestApctaMethodRule
   {
      public static void Main()
      {
          // Indirectly calls DoWork in the full-trust class.
          ClassRequiringFullTrust testClass = new ClassRequiringFullTrust();
          testClass.Access();
      }
   }
}
```

This example produces the following output:

```
Demand for full trust:Request failed.
ClassRequiringFullTrust.DoWork was called.
```

Related rules

- CA2117: APTCA types should only extend APTCA base types

See also

- Secure Coding Guidelines
- Using Libraries from Partially Trusted Code
- Link Demands
- Data and Modeling

CA2117: APTCA types should only extend APTCA base types

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2117
Category	Microsoft.Security
Breaking change	Breaking

Cause

A public or protected type in an assembly with the System.Security.AllowPartiallyTrustedCallersAttribute attribute inherits from a type declared in an assembly that does not have the attribute.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

By default, public or protected types in assemblies with strong names are implicitly protected by an InheritanceDemand for full trust. Strong-named assemblies marked with the AllowPartiallyTrustedCallersAttribute (APTCA) attribute do not have this protection. The attribute disables the inheritance demand. Exposed types declared in an assembly without an inheritance demand are inheritable by types that do not have full trust.

When the APTCA attribute is present on a fully trusted assembly, and a type in the assembly inherits from a type that does not allow partially trusted callers, a security exploit is possible. If two types `T1` and `T2` meet the following conditions, malicious callers can use the type `T1` to bypass the implicit full trust inheritance demand that protects `T2`:

- `T1` is a public type declared in a fully trusted assembly that has the APTCA attribute.
- `T1` inherits from a type `T2` outside its assembly.
- `T2`'s assembly does not have the APTCA attribute and, therefore, should not be inheritable by types in partially trusted assemblies.

A partially trusted type `X` can inherit from `T1`, which gives it access to inherited members declared in `T2`. Because `T2` does not have the APTCA attribute, its immediate derived type (`T1`) must satisfy an inheritance demand for full trust; `T1` has full trust and therefore satisfies this check. The security risk is because `X` does not participate in satisfying the inheritance demand that protects `T2` from untrusted subclassing. For this reason, types with the APTCA attribute must not extend types that do not have the attribute.

Another security issue, and perhaps a more common one, is that the derived type (`T1`) can, through programmer error, expose protected members from the type that requires full trust (`T2`). When this exposure

occurs, untrusted callers gain access to information that should be available only to fully trusted types.

How to fix violations

If the type reported by the violation is in an assembly that does not require the APTCA attribute, remove it.

If the APTCA attribute is required, add an inheritance demand for full trust to the type. The inheritance demand protects against inheritance by untrusted types.

It is possible to fix a violation by adding the APTCA attribute to the assemblies of the base types reported by the violation. Do not do this without first conducting an intensive security review of all code in the assemblies and all code that depends on the assemblies.

When to suppress warnings

To safely suppress a warning from this rule, you must ensure that protected members exposed by your type do not directly or indirectly allow untrusted callers to access sensitive information, operations, or resources that can be used in a destructive manner.

Example

The following example uses two assemblies and a test application to illustrate the security vulnerability detected by this rule. The first assembly does not have the APTCA attribute and should not be inheritable by partially trusted types (represented by `T2` in the previous discussion).

```
using System;
using System.Security;
using System.Security.Permissions;
using System.Reflection;

// This code is compiled into a strong-named assembly
// that requires full trust.

namespace AptcaTestLibrary
{
   public class ClassRequiringFullTrustWhenInherited
   {
      // This field should be overridable by fully trusted derived types.
      protected static string location = "shady glen";

      // A trusted type can see the data, but cannot change it.
      public virtual string TrustedLocation
      {
         get
         {
            return location;
         }
      }
   }
}
```

The second assembly, represented by `T1` in the previous discussion, is fully trusted and allows partially trusted callers.

```
using System;
using System.Security;
using System.Security.Permissions;
using System.Reflection;

// This class is compiled into an assembly that executes with full
// trust and allows partially trusted callers.

// Violates rule: AptcaTypesShouldOnlyExtendAptcaBaseTypes.

namespace AptcaTestLibrary
{
   public class InheritAClassRequiringFullTrust:
      ClassRequiringFullTrustWhenInherited
   {
      private DateTime meetingDay = DateTime.Parse("February 22 2003");

      public override string ToString()
      {
         // Another error:
         // This method gives untrusted callers the value
         // of TrustedLocation. This information should
         // only be seen by trusted callers.
         string s = String.Format(
            "Meet at the {0} {1}!",
            this.TrustedLocation, meetingDay.ToString());
         return s;
      }
   }
}
```

The test type, represented by `x` in the previous discussion, is in a partially trusted assembly.

```
using System;
using AptcaTestLibrary;

// If this test application is run from the local machine,
//  it gets full trust by default.
// Remove full trust.
[assembly: System.Security.Permissions.PermissionSetAttribute(
   System.Security.Permissions.SecurityAction.RequestRefuse, Name = "FullTrust")]

namespace TestSecLibrary
{
    class InheritFromAFullTrustDecendent : ClassRequiringFullTrust
    {
        public InheritFromAFullTrustDecendent()
        {
            // This constructor maliciously overwrites the protected
            // static member in the fully trusted class.
            // Trusted types will now get the wrong information from
            // the TrustedLocation property.
            InheritFromAFullTrustDecendent.location = "sunny meadow";
        }

        public override string ToString()
        {
            return InheritFromAFullTrustDecendent.location;
        }
    }

    class TestApctaInheritRule
    {
        public static void Main()
        {
            ClassRequiringFullTrust iclass =
               new ClassRequiringFullTrust();
            Console.WriteLine(iclass.ToString());

            // You cannot create a type that inherits from the full trust type
            // directly, but you can create a type that inherits from
            // the APTCA type which in turn inherits from the full trust type.

            InheritFromAFullTrustDecendent inherit =
               new InheritFromAFullTrustDecendent();
            //Show the inherited protected member has changed.
            Console.WriteLine("From Test: {0}", inherit.ToString());

            // Trusted types now get the wrong information from
            // the TrustedLocation property.
            Console.WriteLine(iclass.ToString());
        }
    }
}
```

This example produces the following output:

```
Meet at the shady glen 2/22/2003 12:00:00 AM!
From Test: sunny meadow
Meet at the sunny meadow 2/22/2003 12:00:00 AM!
```

Related rules

CA2116: APTCA methods should only call APTCA methods

See also

- Secure Coding Guidelines
- Using Libraries from Partially Trusted Code

CA2118: Review SuppressUnmanagedCodeSecurityAttribute usage

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2118
Category	Microsoft.Security
Breaking change	Breaking

Cause

A public or protected type or member has the System.Security.SuppressUnmanagedCodeSecurityAttribute attribute.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

SuppressUnmanagedCodeSecurityAttribute changes the default security system behavior for members that execute unmanaged code using COM interop or platform invocation. Generally, the system makes a Data and Modeling for unmanaged code permission. This demand occurs at run time for every invocation of the member, and checks every caller in the call stack for permission. When the attribute is present, the system makes a Link Demands for the permission: the permissions of the immediate caller are checked when the caller is JIT-compiled.

This attribute is primarily used to increase performance; however, the performance gains come with significant security risks. If you place the attribute on public members that call native methods, the callers in the call stack (other than the immediate caller) do not need unmanaged code permission to execute unmanaged code. Depending on the public member's actions and input handling, it might allow untrustworthy callers to access functionality normally restricted to trustworthy code.

.NET relies on security checks to prevent callers from gaining direct access to the current process's address space. Because this attribute bypasses normal security, your code poses a serious threat if it can be used to read or write to the process's memory. Note that the risk is not limited to methods that intentionally provide access to process memory; it is also present in any scenario where malicious code can achieve access by any means, for example, by providing surprising, malformed, or invalid input.

The default security policy does not grant unmanaged code permission to an assembly unless it is executing from the local computer or is a member of one of the following groups:

- My Computer Zone Code Group
- Microsoft Strong Name Code Group
- ECMA Strong Name Code Group

How to fix violations

Carefully review your code to ensure that this attribute is absolutely necessary. If you are unfamiliar with managed code security, or do not understand the security implications of using this attribute, remove it from your code. If the attribute is required, you must ensure that callers cannot use your code maliciously. If your code does not have permission to execute unmanaged code, this attribute has no effect and should be removed.

When to suppress warnings

To safely suppress a warning from this rule, you must ensure that your code does not provide callers access to native operations or resources that can be used in a destructive manner.

Example 1

The following example violates the rule.

```
using System.Security;

// These two classes are identical
// except for the location of the attribute.

namespace SecurityRulesLibrary
{
    public class MyBadMemberClass
   {
      [SuppressUnmanagedCodeSecurityAttribute()]
      public void DoWork()
      {
         FormatHardDisk();
      }

      void FormatHardDisk()
      {
         // Code that calls unmanaged code.
      }
   }

   [SuppressUnmanagedCodeSecurityAttribute()]
   public class MyBadTypeClass
   {
      public void DoWork()
      {
         FormatHardDisk();
      }

      void FormatHardDisk()
      {
         // Code that calls unmanaged code.
      }
   }
}
```

Example 2

In the following example, the `DoWork` method provides a publicly accessible code path to the platform invocation method `FormatHardDisk`.

```
using System.Security;
using System.Runtime.InteropServices;

namespace SecurityRulesLibrary
{
   public class SuppressIsOnPlatformInvoke
   {
      // The DoWork method is public and provides unsecured access
      // to the platform invoke method FormatHardDisk.
      [SuppressUnmanagedCodeSecurityAttribute()]
      [DllImport("native.dll")]

      private static extern void FormatHardDisk();
      public void DoWork()
      {
         FormatHardDisk();
      }
   }

   // Having the attribute on the type also violates the rule.
   [SuppressUnmanagedCodeSecurityAttribute()]
   public class SuppressIsOnType
   {
      [DllImport("native.dll")]

      private static extern void FormatHardDisk();
      public void DoWork()
      {
         FormatHardDisk();
      }
   }
}
```

Example 3

In the following example, the public method `DoDangerousThing` causes a violation. To resolve the violation, `DoDangerousThing` should be made private, and access to it should be through a public method secured by a security demand, as illustrated by the `DoWork` method.

```
using System.Security;
using System.Security.Permissions;
using System.Runtime.InteropServices;

namespace SecurityRulesLibrary
{
   [SuppressUnmanagedCodeSecurityAttribute()]
   public class BadTypeWithPublicPInvokeAndSuppress
   {
      [DllImport("native.dll")]

      public static extern void DoDangerousThing();
      public void DoWork()
      {
         // Note that because DoDangerousThing is public, this
         // security check does not resolve the violation.
         // This only checks callers that go through DoWork().
         SecurityPermission secPerm = new SecurityPermission(
            SecurityPermissionFlag.ControlPolicy |
            SecurityPermissionFlag.ControlEvidence
         );
         secPerm.Demand();
         DoDangerousThing();
      }
   }
}
```

See also

- System.Security.SuppressUnmanagedCodeSecurityAttribute
- Secure Coding Guidelines
- Data and Modeling
- Link Demands

CA2119: Seal methods that satisfy private interfaces

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2119
Category	Microsoft.Security
Breaking change	Breaking

Cause

An inheritable public type provides an overridable method implementation of an `internal` (`Friend` in Visual Basic) interface.

Rule description

Interface methods have public accessibility, which cannot be changed by the implementing type. An internal interface creates a contract that is not intended to be implemented outside the assembly that defines the interface. A public type that implements a method of an internal interface using the `virtual` (`Overridable` in Visual Basic) modifier allows the method to be overridden by a derived type that is outside the assembly. If a second type in the defining assembly calls the method and expects an internal-only contract, behavior might be compromised when, instead, the overridden method in the outside assembly is executed. This creates a security vulnerability.

How to fix violations

To fix a violation of this rule, prevent the method from being overridden outside the assembly by using one of the following:

- Make the declaring type `sealed` (`NotInheritable` in Visual Basic).
- Change the accessibility of the declaring type to `internal` (`Friend` in Visual Basic).
- Remove all public constructors from the declaring type.
- Implement the method without using the `virtual` modifier.
- Implement the method explicitly.

When to suppress warnings

It is safe to suppress a warning from this rule if, after careful review, no security issues exist that might be exploitable if the method is overridden outside the assembly.

Example 1

The following example shows a type, `BaseImplementation`, that violates this rule.

```
using namespace System;

namespace SecurityLibrary
{
   // Internal by default.
   interface class IValidate
   {
      bool UserIsValidated();
   };

   public ref class BaseImplementation : public IValidate
   {
   public:
      virtual bool UserIsValidated()
      {
         return false;
      }
   };

   public ref class UseBaseImplementation
   {
   public:
      void SecurityDecision(BaseImplementation^ someImplementation)
      {
         if(someImplementation->UserIsValidated() == true)
         {
            Console::WriteLine("Account number & balance.");
         }
         else
         {
            Console::WriteLine("Please login.");
         }
      }
   };
}
```

```
using System;

namespace SecurityLibrary
{
   // Internal by default.
   interface IValidate
   {
      bool UserIsValidated();
   }

   public class BaseImplementation : IValidate
   {
      public virtual bool UserIsValidated()
      {
         return false;
      }
   }

   public class UseBaseImplementation
   {
      public void SecurityDecision(BaseImplementation someImplementation)
      {
         if(someImplementation.UserIsValidated() == true)
         {
            Console.WriteLine("Account number & balance.");
         }
         else
         {
            Console.WriteLine("Please login.");
         }
      }
   }
}
```

```
Imports System

Namespace SecurityLibrary

   Interface IValidate
      Function UserIsValidated() As Boolean
   End Interface

   Public Class BaseImplementation
      Implements IValidate

      Overridable Function UserIsValidated() As Boolean _
         Implements IValidate.UserIsValidated
         Return False
      End Function

   End Class

   Public Class UseBaseImplementation

      Sub SecurityDecision(someImplementation As BaseImplementation)

         If(someImplementation.UserIsValidated() = True)
            Console.WriteLine("Account number & balance.")
         Else
            Console.WriteLine("Please login.")
         End If

      End Sub

   End Class

End Namespace
```

Example 2

The following example exploits the virtual method implementation of the previous example.

```
using namespace System;

namespace SecurityLibrary
{
   public ref class BaseImplementation
   {
   public:
      virtual bool UserIsValidated()
      {
         return false;
      }
   };

   public ref class UseBaseImplementation
   {
   public:
      void SecurityDecision(BaseImplementation^ someImplementation)
      {
         if(someImplementation->UserIsValidated() == true)
         {
            Console::WriteLine("Account number & balance.");
         }
         else
         {
            Console::WriteLine("Please login.");
         }
      }
   };
}
```

```
using System;

namespace SecurityLibrary
{
    public class BaseImplementation
    {
        public virtual bool UserIsValidated()
        {
            return false;
        }
    }

    public class UseBaseImplementation
    {
        public void SecurityDecision(BaseImplementation someImplementation)
        {
            if (someImplementation.UserIsValidated() == true)
            {
                Console.WriteLine("Account number & balance.");
            }
            else
            {
                Console.WriteLine("Please login.");
            }
        }
    }
}
```

```
Imports System

Namespace SecurityLibrary

   Public Class BaseImplementation

      Overridable Function UserIsValidated() As Boolean
         Return False
      End Function

   End Class

   Public Class UseBaseImplementation

      Sub SecurityDecision(someImplementation As BaseImplementation)

         If(someImplementation.UserIsValidated() = True)
            Console.WriteLine("Account number & balance.")
         Else
            Console.WriteLine("Please login.")
         End If

      End Sub

   End Class

End Namespace
```

See also

- Interfaces
- Interfaces

CA2120: Secure serialization constructors

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2120
Category	Microsoft.Security
Breaking change	Breaking

Cause

The type implements the System.Runtime.Serialization.ISerializable interface, is not a delegate or interface, and is declared in an assembly that allows partially trusted callers. The type has a constructor that takes a System.Runtime.Serialization.SerializationInfo object and a System.Runtime.Serialization.StreamingContext object (the signature of the serialization constructor). This constructor is not secured by a security check, but one or more of the regular constructors in the type is secured.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule is relevant for types that support custom serialization. A type supports custom serialization if it implements the System.Runtime.Serialization.ISerializable interface. The serialization constructor is required and is used to de-serialize, or re-create objects that have been serialized using the System.Runtime.Serialization.ISerializable.GetObjectData method. Because the serialization constructor allocates and initializes objects, security checks that are present on regular constructors must also be present on the serialization constructor. If you violate this rule, callers that could not otherwise create an instance could use the serialization constructor to do this.

How to fix violations

To fix a violation of this rule, protect the serialization constructor with security demands that are identical to those protecting other constructors.

When to suppress warnings

Do not suppress a violation of the rule.

Example

The following example shows a type that violates the rule.

```
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 SecurityRulesLibrary
{
    [Serializable]
    public class SerializationConstructorsRequireSecurity : ISerializable
    {
        private  int n1;
        // This is a regular constructor secured by a demand.
        [FileIOPermissionAttribute(SecurityAction.Demand, Unrestricted = true)]
        public SerializationConstructorsRequireSecurity ()
        {
           n1 = -1;
        }
        // This is the serialization constructor.
        // Violates rule: SecureSerializationConstructors.
        protected SerializationConstructorsRequireSecurity (SerializationInfo info, StreamingContext
context)
        {
           n1 = (int) info.GetValue("n1", typeof(int));
        }
        void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
        {
           info.AddValue("n1", n1);
        }
    }

 }
```

Related rules

CA2229: Implement serialization constructors

CA2237: Mark ISerializable types with SerializableAttribute

See also

- System.Runtime.Serialization.ISerializable
- System.Runtime.Serialization.SerializationInfo
- System.Runtime.Serialization.StreamingContext

CA2121: Static constructors should be private

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2121
Category	Microsoft.Security
Breaking change	Breaking

Cause

A type has a static constructor that is not private.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A static constructor, also known as a class constructor, is used to initialize a type. The system calls the static constructor before the first instance of the type is created or any static members are referenced. The user has no control over when the static constructor is called. If a static constructor is not private, it can be called by code other than the system. Depending on the operations that are performed in the constructor, this can cause unexpected behavior.

This rule is enforced by the C# and Visual Basic compilers.

How to fix violations

Violations are typically caused by one of the following actions:

- You defined a static constructor for your type and did not make it private.
- The programming language compiler added a default static constructor to your type and did not make it private.

To fix the first kind of violation, make your static constructor private. To fix the second kind, add a private static constructor to your type.

When to suppress warnings

Do not suppress these violations. If your software design requires an explicit call to a static constructor, it is likely that the design contains serious flaws and should be reviewed.

CA2122: Do not indirectly expose methods with link demands

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2122
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A public or protected member has a Link Demands and is called by a member that does not perform any security checks.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A link demand checks the permissions of the immediate caller only. If a member `x` makes no security demands of its callers, and calls code protected by a link demand, a caller without the necessary permission can use `x` to access the protected member.

How to fix violations

Add a security Data and Modeling or link demand to the member so that it no longer provides unsecured access to the link demand-protected member.

When to suppress warnings

To safely suppress a warning from this rule, you must make sure that your code does not grant its callers access to operations or resources that can be used in a destructive manner.

Example 1

The following examples show a library that violates the rule, and an application that demonstrates the library's weakness. The sample library provides two methods that together violate the rule. The `EnvironmentSetting` method is secured by a link demand for unrestricted access to environment variables. The `DomainInformation` method makes no security demands of its callers before it calls `EnvironmentSetting`.

```
using System;
using System.IO;
using System.Security;
using System.Security.Permissions;

namespace SecurityRulesLibrary
{
   public class DoNotIndirectlyExposeMethodsWithLinkDemands
   {
      // Violates rule: DoNotIndirectlyExposeMethodsWithLinkDemands.
      public static string DomainInformation()
      {
         return EnvironmentSetting("USERDNSDOMAIN");
      }

      // Library method with link demand.
      // This method holds its immediate callers responsible for securing the information.
      // Because a caller must have unrestricted permission, the method asserts read permission
      // in case some caller in the stack does not have this permission.

      [EnvironmentPermissionAttribute(SecurityAction.LinkDemand, Unrestricted=true)]
      public static string EnvironmentSetting(string environmentVariable)
      {
         EnvironmentPermission envPermission = new EnvironmentPermission(
EnvironmentPermissionAccess.Read,environmentVariable);
         envPermission.Assert();

         return Environment.GetEnvironmentVariable(environmentVariable);
      }
   }
}
```

Example 2

The following application calls the unsecured library member.

```
using System;
using SecurityRulesLibrary;
using System.Security;
using System.Security.Permissions;

// You have no permission to access the sensitive information,
// but you will get data from the unprotected method.
[assembly:EnvironmentPermissionAttribute(
   SecurityAction.RequestRefuse,Unrestricted=true)]
namespace TestUnsecuredMembers
{
   class TestUnsecured
   {
      [STAThread]
      static void Main(string[] args)
      {
         string value = null;
         try
         {
            value = DoNotIndirectlyExposeMethodsWithLinkDemands.DomainInformation();
         }
         catch (SecurityException e)
         {
            Console.WriteLine(
               "Call to unsecured member was stopped by code access security! {0}",
               e.Message);
            throw;
         }
         if (value != null)
         {
            Console.WriteLine("Value from unsecured member: {0}", value);
         }
      }
   }
}
```

This example produces the following output:

```
*Value from unsecured member: seattle.corp.contoso.com
```

See also

- Secure Coding Guidelines
- Link Demands
- Data and Modeling

CA2123: Override link demands should be identical to base

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2123
Category	Microsoft.Security
Breaking change	Breaking

Cause

A public or protected method in a public type overrides a method or implements an interface, and does not have the same Link Demands as the interface or virtual method.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule matches a method to its base method, which is either an interface or a virtual method in another type, and then compares the link demands on each. A violation is reported if either the method or the base method has a link demand and the other does not.

If this rule is violated, a malicious caller can bypass the link demand merely by calling the unsecured method.

How to fix violations

To fix a violation of this rule, apply the same link demand to the override method or implementation. If this is not possible, mark the method with a full demand or remove the attribute altogether.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows various violations of this rule.

```
using System.Security;
using System.Security.Permissions;
using System;

namespace SecurityRulesLibrary
{
   public interface ITestOverrides
   {
```

```
      [EnvironmentPermissionAttribute(SecurityAction.LinkDemand, Unrestricted=true)]
      Object GetFormat(Type formatType);
   }

   public class OverridesAndSecurity : ITestOverrides
   {
      // Rule violation: The interface has security, and this implementation does not.
      object ITestOverrides.GetFormat(Type formatType)
      {
         return (formatType == typeof(OverridesAndSecurity) ? this : null);
      }

      // These two methods are overridden by DerivedClass and DoublyDerivedClass.
      [EnvironmentPermissionAttribute(SecurityAction.LinkDemand, Unrestricted=true)]
      public virtual void DoSomething()
      {
         Console.WriteLine("Doing something.");
      }

      public virtual void DoSomethingElse()
      {
         Console.WriteLine("Doing some other thing.");
      }
   }

   public class DerivedClass : OverridesAndSecurity, ITestOverrides
   {
      //  Rule violation: The interface has security, and this implementation does not.
      public object GetFormat(Type formatType)
      {
         return (formatType == typeof(OverridesAndSecurity) ? this : null);
      }

      // Rule violation: This does not have security, but the base class version does.
      public override void DoSomething()
      {
         Console.WriteLine("Doing some derived thing.");
      }

      // Rule violation: This has security, but the base class version does not.
      [EnvironmentPermissionAttribute(SecurityAction.LinkDemand, Unrestricted=true)]
      public override void DoSomethingElse()
      {
         Console.WriteLine("Doing some other derived thing.");
      }
   }

   public class DoublyDerivedClass : DerivedClass
   {
      // The OverridesAndSecurity version of this method does not have security.
      // Base class DerivedClass's version does.
      // The DoublyDerivedClass version does not violate the rule, but the
      // DerivedClass version does violate the rule.
      public override void DoSomethingElse()
      {
         Console.WriteLine("Doing some other derived thing.");
      }
   }
}
```

See also

- Secure Coding Guidelines
- Link Demands

CA2124: Wrap vulnerable finally clauses in outer try

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2124
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

In versions 1.0 and 1.1 of the .NET Framework, a public or protected method contains a `try` / `catch` / `finally` block. The `finally` block appears to reset security state and is not enclosed in a `finally` block.

> **NOTE**
>
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule locates `try` / `finally` blocks in code that targets versions 1.0 and 1.1 of the .NET Framework that might be vulnerable to malicious exception filters present in the call stack. If sensitive operations such as impersonation occur in the try block, and an exception is thrown, the filter can execute before the `finally` block. For the impersonation example, this means that the filter would execute as the impersonated user. Filters are currently implementable only in Visual Basic.

> **NOTE**
>
> In versions 2.0 and later of the .NET Framework, the runtime automatically protects a `try` / `catch` / `finally` block from malicious exception filters, if the reset occurs directly within the method that contains the exception block.

How to fix violations

Place the unwrapped `try` / `finally` in an outer try block. See the second example that follows. This forces the `finally` to execute before filter code.

When to suppress warnings

Do not suppress a warning from this rule.

Pseudo-code example

Description

The following pseudo-code illustrates the pattern detected by this rule.

```
try {
   // Do some work.
   Impersonator imp = new Impersonator("John Doe");
   imp.AddToCreditCardBalance(100);
}
finally {
   // Reset security state.
   imp.Revert();
}
```

The following pseudo-code shows the pattern that you can use to protect your code and satisfy this rule.

```
try {
     try {
        // Do some work.
     }
     finally {
        // Reset security state.
     }
}
catch()
{
    throw;
}
```

CA2126: Type link demands require inheritance demands

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2126
Category	Microsoft.Security
Breaking change	Breaking

Cause

A public unsealed type is protected with a link demand, has an overridable method, and neither the type nor the method is protected with an inheritance demand.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A link demand on a method or its declaring type requires the immediate caller of the method to have the specified permission. An inheritance demand on a method requires an overriding method to have the specified permission. An inheritance demand on a type requires a deriving class to have the specified permission.

How to fix violations

To fix a violation of this rule, secure the type or the method with an inheritance demand for the same permission as the link demand.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a type that violates the rule.

```
using namespace System;
using namespace System::Security::Permissions;

namespace SecurityLibrary
{
    [EnvironmentPermission(SecurityAction::LinkDemand, Read = "PATH")]
    public ref class TypesWithLinkDemands
    {
    protected:
        virtual void UnsecuredMethod() {}

        [EnvironmentPermission(SecurityAction::InheritanceDemand,
           Read = "PATH")]
        virtual void SecuredMethod() {}
    };
}
```

```
Imports System
Imports System.Security.Permissions

Namespace SecurityLibrary

    <EnvironmentPermission(SecurityAction.LinkDemand, Read:="PATH")> _
    Public Class TypesWithLinkDemands

        Protected Overridable Sub UnsecuredMethod()
        End Sub

        <EnvironmentPermission(SecurityAction.InheritanceDemand, Read:="PATH")> _
        Protected Overridable Sub SecuredMethod()
        End Sub

    End Class

End Namespace
```

```
using System;
using System.Security.Permissions;

namespace SecurityLibrary
{
   [EnvironmentPermission(SecurityAction.LinkDemand, Read = "PATH")]
   public class TypesWithLinkDemands
   {
      public virtual void UnsecuredMethod() {}

      [EnvironmentPermission(SecurityAction.InheritanceDemand, Read = "PATH")]
      public virtual void SecuredMethod() { }
   }
}
```

Related rules

CA2108: Review declarative security on value types

CA2112: Secured types should not expose fields

CA2122: Do not indirectly expose methods with link demands

CA2123: Override link demands should be identical to base

See also

- Secure Coding Guidelines
- Link Demands

CA2130: Security critical constants should be transparent

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2130
Category	Microsoft.Security
Breaking change	Breaking

Cause

A constant field or an enumeration member is marked with the SecurityCriticalAttribute.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Transparency enforcement is not enforced for constant values because compilers inline constant values so that no lookup is required at run time. Constant fields should be security transparent so that code reviewers do not assume that transparent code cannot access the constant.

How to fix violations

To fix a violation of this rule, remove the SecurityCritical attribute from the field or value.

When to suppress warnings

Do not suppress a warning from this rule.

Example

In the following examples, the enum value `EnumWithCriticalValues.CriticalEnumValue` and the constant `CriticalConstant` raise this warning. To fix the issues, remove the [`SecurityCritical`] attribute to make them security transparent.

```
using System;
using System.Security;

//[assembly: SecurityRules(SecurityRuleSet.Level2)]
//[assembly: AllowPartiallyTrustedCallers]

namespace TransparencyWarningsDemo
{

    public enum EnumWithCriticalValues
    {
        TransparentEnumValue,

        // CA2130 violation
        [SecurityCritical]
        CriticalEnumValue
    }

    public class ClassWithCriticalConstant
    {
        // CA2130 violation
        [SecurityCritical]
        public const int CriticalConstant = 21;
    }
}
```

CA2131: Security critical types may not participate in type equivalence

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2131
Category	Microsoft.Security
Breaking change	Breaking

Cause

A type participates in type equivalence and a either the type itself, or a member or field of the type, is marked with the SecurityCriticalAttribute attribute.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule fires on any critical types or types that contain critical methods or fields that are participating in type equivalence. When the CLR detects such a type, it fails to load it with a TypeLoadException at run time. Typically, this rule fires only when users implement type equivalence manually rather than by relying on tlbimp and the compilers to do the type equivalence.

How to fix violations

To fix a violation of this rule, remove the SecurityCritical attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following examples demonstrate an interface, a method, and a field that will cause this rule to fire.

```
using System;
using System.Security;
using System.Runtime.InteropServices;

[assembly: SecurityRules(SecurityRuleSet.Level2)]
[assembly: AllowPartiallyTrustedCallers]

namespace TransparencyWarningsDemo
{

    // CA2131 error - critical type participating in equivilance
    [SecurityCritical]
    [TypeIdentifier("3a5b6203-2bf1-4f83-b5b4-1bdc334ad3ea", "ICriticalEquivilentInterface")]
    public interface ICriticalEquivilentInterface
    {
        void Method1();
    }

    [TypeIdentifier("3a5b6203-2bf1-4f83-b5b4-1bdc334ad3ea", "ITransparentEquivilentInterface")]
    public interface ITransparentEquivilentInterface
    {
        // CA2131 error - critical method in a type participating in equivilance
        [SecurityCritical]
        void CriticalMethod();
    }

    [SecurityCritical]
    [TypeIdentifier("3a5b6203-2bf1-4f83-b5b4-1bdc334ad3ea", "ICriticalEquivilentInterface")]
    public struct EquivilentStruct
    {
        // CA2131 error - critical field in a type participating in equivalence
        [SecurityCritical]
        public int CriticalField;
    }
}
```

See also

Security-Transparent Code, Level 2

CA2132: Default constructors must be at least as critical as base type default constructors

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2132
Category	Microsoft.Security
Breaking change	Breaking

> **NOTE**
> This warning is only applied to code that is running the CoreCLR (the version of the CLR that is specific to Silverlight web applications).

Cause

The transparency attribute of the default constructor of a derived class is not as critical as the transparency of the base class.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Types and members that have the SecurityCriticalAttribute cannot be used by Silverlight application code. Security-critical types and members can be used only by trusted code in the .NET Framework for Silverlight class library. Because a public or protected construction in a derived class must have the same or greater transparency than its base class, a class in an application cannot be derived from a class marked SecurityCritical.

For CoreCLR platform code, if a base type has a public or protected non-transparent default constructor then the derived type must obey the default constructor inheritance rules. The derived type must also have a default constructor and that constructor must be at least as critical default constructor of the base type.

How to fix violations

To fix the violation, remove the type or do not derive from security non-transparent type.

When to suppress warnings

Do not suppress warnings from this rule. Violations of this rule by application code will result in the CoreCLR refusing to load the type with a TypeLoadException.

Code

```
using System;
using System.Security;

namespace TransparencyWarningsDemo
{

    public class BaseWithSafeCriticalDefaultCtor
    {
        [SecuritySafeCritical]
        public BaseWithSafeCriticalDefaultCtor() { }
    }

    public class DerivedWithNoDefaultCtor : BaseWithSafeCriticalDefaultCtor
    {
        // CA2132 violation - since the base has a public or protected non-transparent default .ctor, the
        // derived type must also have a default .ctor
    }

    public class DerivedWithTransparentDefaultCtor : BaseWithSafeCriticalDefaultCtor
    {
        // CA2132 violation - since the base has a safe critical default .ctor, the derived type must have
        // either a safe critical or critical default .ctor.  This is fixed by making this .ctor safe
critical
        // (however, user code cannot be safe critical, so this fix is platform code only).
        DerivedWithTransparentDefaultCtor() { }
    }

    public class BaseWithCriticalCtor
    {
        [SecurityCritical]
        public BaseWithCriticalCtor() { }
    }

    public class DerivedWithSafeCriticalDefaultCtor : BaseWithSafeCriticalDefaultCtor
    {
        // CA2132 violation - since the base has a critical default .ctor, the derived must also have a
critical
        // default .ctor.  This is fixed by making this .ctor critical, which is not available to user code
        [SecuritySafeCritical]
        public DerivedWithSafeCriticalDefaultCtor() { }
    }
}
```

CA2133: Delegates must bind to methods with consistent transparency

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2133
Category	Microsoft.Security
Breaking change	Breaking

> **NOTE**
> This warning is only applied to code that is running the CoreCLR (the version of the CLR that is specific to Silverlight web applications).

Cause

This warning fires on a method that binds a delegate that is marked with the SecurityCriticalAttribute to a method that is transparent or that is marked with the SecuritySafeCriticalAttribute. The warning also fires a method that binds a delegate that is transparent or safe-critical to a critical method.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Delegate types and the methods that they bind to must have consistent transparency. Transparent and safe-critical delegates may only bind to other transparent or safe-critical methods. Similarly, critical delegates may only bind to critical methods. These binding rules ensure that the only code that can invoke a method via a delegate could have also invoked the same method directly. For example, binding rules prevent transparent code from calling critical code directly via a transparent delegate.

How to fix violations

To fix a violation of this warning, change the transparency of the delegate or of the method that it binds so that the transparency of the two are equivalent.

When to suppress warnings

Do not suppress a warning from this rule.

Code

```
using System;
using System.Security;

namespace TransparencyWarningsDemo
{

    public delegate void TransparentDelegate();

    [SecurityCritical]
    public delegate void CriticalDelegate();

    public class TransparentType
    {
        void DelegateBinder()
        {
            // CA2133 violation - binding a transparent delegate to a critical method
            TransparentDelegate td = new TransparentDelegate(CriticalTarget);

            // CA2133 violation - binding a critical delegate to a transparent method
            CriticalDelegate cd = new CriticalDelegate(TransparentTarget);
        }

        [SecurityCritical]
        void CriticalTarget() { }

        void TransparentTarget() { }
    }
}
```

CA2134: Methods must keep consistent transparency when overriding base methods

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2134
Category	Microsoft.Security
Breaking change	Breaking

Cause

This rule fires when a method marked with the SecurityCriticalAttribute overrides a method that is transparent or marked with the SecuritySafeCriticalAttribute. The rule also fires when a method that is transparent or marked with the SecuritySafeCriticalAttribute overrides a method that is marked with a SecurityCriticalAttribute.

The rule is applied when overriding a virtual method or implementing an interface.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule fires on attempts to change the security accessibility of a method further up the inheritance chain. For example, if a virtual method in a base class is transparent or safe-critical, then the derived class must override it with a transparent or safe-critical method. Conversely, if the virtual is security critical, the derived class must override it with a security critical method. The same rule applies for implementing interface methods.

Transparency rules are enforced when the code is JIT compiled instead of at run time, so that the transparency calculation does not have dynamic type information. Therefore, the result of the transparency calculation must be able to be determined solely from the static types being JIT-compiled, regardless of the dynamic type.

How to fix violations

To fix a violation of this rule, change the transparency of the method that is overriding a virtual method or implementing an interface to match the transparency of the virtual or interface method.

When to suppress warnings

Do not suppress warnings from this rule. Violations of this rule result in a run-time TypeLoadException for assemblies that use level 2 transparency.

Examples

Code

```
using System;
using System.Security;

namespace TransparencyWarningsDemo
{

    public interface IInterface
    {
        void TransparentInterfaceMethod();

        [SecurityCritical]
        void CriticalInterfaceMethod();
    }

    public class Base
    {
        public virtual void TransparentVirtual() { }

        [SecurityCritical]
        public virtual void CriticalVirtual() { }
    }

    public class Derived : Base, IInterface
    {
        // CA2134 violation - implementing a transparent method with a critical one.  This can be fixed by
any of:
        //   1. Making IInterface.TransparentInterfaceMethod security critical
        //   2. Making Derived.TransparentInterfaceMethod transparent
        //   3. Making Derived.TransparentInterfaceMethod safe critical
        [SecurityCritical]
        public void TransparentInterfaceMethod() { }

        // CA2134 violation - implementing a critical method with a transparent one.  This can be fixed by
any of:
        //   1. Making IInterface.CriticalInterfaceMethod transparent
        //   2. Making IInterface.CriticalInterfaceMethod safe critical
        //   3. Making Derived.TransparentInterfaceMethod critical
        public void CriticalInterfaceMethod() { }

        // CA2134 violation - overriding a transparent method with a critical one.  This can be fixed by any
of:
        //   1. Making Base.TrasnparentVirtual critical
        //   2. Making Derived.TransparentVirtual transparent
        //   3. Making Derived.TransparentVirtual safe critical
        [SecurityCritical]
        public override void TransparentVirtual() { }

        // CA2134 violation - overriding a critical method with a transparent one.  This can be fixed by any
of:
        //   1. Making Base.CriticalVirtual transparent
        //   2. Making Base.CriticalVirtual safe critical
        //   3. Making Derived.CriticalVirtual critical
        public override void CriticalVirtual() { }
    }

}
```

See also

Security-Transparent Code, Level 2

CA2135: Level 2 assemblies should not contain LinkDemands

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2135
Category	Microsoft.Security
Breaking change	Breaking

Cause

A class or class member is using a SecurityAction in an application that is using Level 2 security.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

LinkDemands are deprecated in the level 2 security rule set. Instead of using LinkDemands to enforce security at just-in-time (JIT) compilation time, mark the methods, types, and fields with the SecurityCriticalAttribute attribute.

How to fix violations

To fix a violation of this rule, remove the SecurityAction and mark the type or member with the SecurityCriticalAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Example

In the following example, the SecurityAction should be removed and the method marked with the SecurityCriticalAttribute attribute.

```
using System;
using System.Security;
using System.Security.Permissions;

namespace TransparencyWarningsDemo
{

    public class MethodsProtectedWithLinkDemandsClass
    {
        // CA2135 violation - the LinkDemand should be removed, and the method marked [SecurityCritical]
instead
        [SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
        public void ProtectedMethod()
        {
        }
    }
}
```

CA2136: Members should not have conflicting transparency annotations

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2136
Category	Microsoft.Security
Breaking change	Breaking

Cause

This rule fires 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.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

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.

How to fix violations

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.

When to suppress warnings

Do not suppress warnings from this rule.

Example

In the following example, a method is marked with the SecuritySafeCriticalAttribute attribute and it is a member of a class that is marked with the SecurityCriticalAttribute attribute. The security safe attribute should be removed.

```
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]
        public void SafeCriticalMethod()
        {
        }
    }
}
```

CA2137: Transparent methods must contain only verifiable IL

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2137
Category	Microsoft.Security
Breaking change	Breaking

Cause

A method contains unverifiable code or returns a type by reference.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule fires on attempts by security transparent code to execute unverifiable MSIL (Microsoft Intermediate Language). However, the rule does not contain a full IL verifier, and instead uses heuristics to catch most violations of MSIL verification.

To be certain that your code contains only verifiable MSIL, run Peverify.exe (PEVerify Tool) on your assembly. Run PEVerify with the **/transparent** option which limits the output to only unverifiable transparent methods which would cause an error. If the /transparent option is not used, PEVerify also verifies critical methods that are allowed to contain unverifiable code.

How to fix violations

To fix a violation of this rule, mark the method with the SecurityCriticalAttribute or SecuritySafeCriticalAttribute attribute, or remove the unverifiable code.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The method in this example uses unverifiable code and should be marked with the SecurityCriticalAttribute or SecuritySafeCriticalAttribute attribute.

```
using System;
using System.Security;


namespace TransparencyWarningsDemo
{

    public class UnverifiableMethodClass
    {
        // CA2137 violation - transparent method with unverifiable code.  This method should become critical
or
        // safe critical
    //    public unsafe byte[] UnverifiableMethod(int length)
    //    {
    //        byte[] bytes = new byte[length];
    //        fixed (byte* pb = bytes)
    //        {
    //            *pb = (byte)length;
    //        }

    //        return bytes;
    //    }
    }

}
```

CA2138: Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2138
Category	Microsoft.Security
Breaking change	Breaking

Cause

A security transparent method calls a method that is marked with the SuppressUnmanagedCodeSecurityAttribute attribute.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule fires on any transparent method that calls directly into native code, for example, by using a P/Invoke (platform invoke) call. P/Invoke and COM interop methods that are marked with the SuppressUnmanagedCodeSecurityAttribute attribute result in a LinkDemand being done against the calling method. Because security transparent code cannot satisfy LinkDemands, the code also cannot call methods that are marked with the SuppressUnmanagedCodeSecurity attribute, or methods of class that is marked with SuppressUnmanagedCodeSecurity attribute. The method will fail, or the demand will be converted to a full demand.

Violations of this rule lead to a MethodAccessException in the Level 2 security transparency model, and a full demand for UnmanagedCode in the Level 1 transparency model.

How to fix violations

To fix a violation of this rule, remove the SuppressUnmanagedCodeSecurityAttribute attribute and mark the method with the SecurityCriticalAttribute or the SecuritySafeCriticalAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Example

```
using System;
using System.Runtime.InteropServices;
using System.Security;


namespace TransparencyWarningsDemo
{

    public class CallSuppressUnmanagedCodeSecurityClass
    {
        [SuppressUnmanagedCodeSecurity]
        [DllImport("kernel32.dll", SetLastError = true)]
        [return: MarshalAs(UnmanagedType.Bool)]
        static extern bool Beep(uint dwFreq, uint dwDuration);

        public void CallNativeMethod()
        {
            // CA2138 violation - transparent method calling a method marked with
SuppressUnmanagedCodeSecurity
            // (this is also a CA2149 violation as well, since this is a P/Invoke and not an interface
call).
            Beep(10000, 1);
        }
    }

}
```

CA2139: Transparent methods may not use the HandleProcessCorruptingExceptions attribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2139
Category	Microsoft.Security
Breaking change	Breaking

Cause

A transparent method is marked with the HandleProcessCorruptedStateExceptionsAttribute attribute.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule fires any method which is transparent and attempts to handle a process corrupting exception by using the HandleProcessCorruptedStateExceptionsAttribute attribute. A process corrupting exception is a CLR version 4.0 exception classification of exceptions such AccessViolationException. The HandleProcessCorruptedStateExceptionsAttribute attribute may only be used by security critical methods, and will be ignored if it is applied to a transparent method. To handle process corrupting exceptions, this method must become security critical or security safe-critical.

How to fix violations

To fix a violation of this rule, remove the HandleProcessCorruptedStateExceptionsAttribute attribute, or mark the method with the SecurityCriticalAttribute or the SecuritySafeCriticalAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Example

In this example, a transparent method is marked with the HandleProcessCorruptedStateExceptionsAttribute attribute and will fail the rule. The method should also be marked with the SecurityCriticalAttribute or the SecuritySafeCriticalAttribute attribute.

```
using System;
using System.Runtime.InteropServices;
using System.Runtime.ExceptionServices;
using System.Security;

namespace TransparencyWarningsDemo
{

    public class HandleProcessCorruptedStateExceptionClass
    {
        [DllImport("SomeModule.dll")]
        private static extern void NativeCode();

        // CA2139 violation - transparent method attempting to handle a process corrupting exception
        [HandleProcessCorruptedStateExceptions]
        public void HandleCorruptingExceptions()
        {
            try
            {
                NativeCode();
            }
            catch (AccessViolationException) { }
        }
    }

}
```

CA2140: Transparent code must not reference security critical items

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2140
Category	Microsoft.Security
Breaking change	Breaking

Cause

A transparent method:

- handles a security critical security exception type
- has a parameter that is marked as a security critical type
- has a generic parameter with a security critical constraints
- has a local variable of a security critical type
- references a type that is marked as security critical
- calls a method that is marked as security critical
- references a field that is marked as security critical
- returns a type that is marked as security critical

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A code element that is marked with the SecurityCriticalAttribute attribute is security critical. A transparent method cannot use a security critical element. If a transparent type attempts to use a security critical type a TypeAccessException, MethodAccessException , or FieldAccessException is raised.

How to fix violations

To fix a violation of this rule, do one of the following:

- Mark the code element that uses the security critical code with the SecurityCriticalAttribute attribute

  - or -

- Remove the SecurityCriticalAttribute attribute from the code elements that are marked as security critical and instead mark them with the SecuritySafeCriticalAttribute or SecurityTransparentAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Example

In the following examples, a transparent method attempts to reference a security critical generic collection, a security critical field, and a security critical method.

```
using System;
using System.Security;
using System.Collections.Generic;

namespace TransparencyWarningsDemo
{

    [SecurityCritical]
    public class SecurityCriticalClass { }

    public class TransparentMethodsReferenceCriticalCodeClass
    {
        [SecurityCritical]
        private object m_criticalField;

        [SecurityCritical]
        private void CriticalMethod() { }

        public void TransparentMethod()
        {
            // CA2140 violation - transparent method accessing a critical type.  This can be fixed by any
of:
            //  1. Make TransparentMethod critical
            //  2. Make TransparentMethod safe critical
            //  3. Make CriticalClass safe critical
            //  4. Make CriticalClass transparent
            List<SecurityCriticalClass> l = new List<SecurityCriticalClass>();

            // CA2140 violation - transparent method accessing a critical field.  This can be fixed by any
of:
            //  1. Make TransparentMethod critical
            //  2. Make TransparentMethod safe critical
            //  3. Make m_criticalField safe critical
            //  4. Make m_criticalField transparent
            m_criticalField = l;

            // CA2140 violation - transparent method accessing a critical method.  This can be fixed by any
of:
            //  1. Make TransparentMethod critical
            //  2. Make TransparentMethod safe critical
            //  3. Make CriticalMethod safe critical
            //  4. Make CriticalMethod transparent
            CriticalMethod();
        }
    }
}
```

See also

- SecurityTransparentAttribute
- SecurityCriticalAttribute
- SecurityTransparentAttribute
- SecurityTreatAsSafeAttribute

- System.Security

CA2141:Transparent methods must not satisfy LinkDemands

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2141
Category	Microsoft.Security
Breaking change	Breaking

Cause

A security transparent method calls a method in an assembly that is not marked with the AllowPartiallyTrustedCallersAttribute (APTCA) attribute, or a security transparent method satisfies a SecurityAction `.LinkDemand` for a type or a method.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Satisfying a LinkDemand is a security sensitive operation that can cause unintentional elevation of privilege. Security transparent code must not satisfy LinkDemands, because it is not subject to the same security audit requirements as security critical code. Transparent methods in security rule set level 1 assemblies will cause all LinkDemands they satisfy to be converted to full demands at run time, which can cause performance problems. In security rule set level 2 assemblies, transparent methods will fail to compile in the just-in-time (JIT) compiler if they attempt to satisfy a LinkDemand.

In assemblies that use Level 2 security, attempts by a security transparent method to satisfy a LinkDemand or call a method in a non-APTCA assembly raises a MethodAccessException; in Level 1 assemblies the LinkDemand becomes a full Demand.

How to fix violations

To fix a violation of this rule, mark the accessing method with the SecurityCriticalAttribute or SecuritySafeCriticalAttribute attribute, or remove the LinkDemand from the accessed method.

When to suppress warnings

Do not suppress a warning from this rule.

Example

In this example, a transparent method attempts to call a method that has a LinkDemand. This rule will fire on this code.

```
using System;
using System.Security.Permissions;


namespace TransparencyWarningsDemo
{

    public class TransparentMethodSatisfiesLinkDemandsClass
    {
        [SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
        public void LinkDemandMethod() { }


        public void TransparentMethod()
        {
            // CA2141 violation - transparent method calling a method protected with a link demand.  Any of
the
            // following fixes will work here:
            //  1. Make TransparentMethod critical
            //  2. Make TransparentMethod safe critical
            //  3. Remove the LinkDemand from LinkDemandMethod  (In this case, that would be recommended
anyway
            //     since it's level 2 -- however you could imagine it in a level 1 assembly)
            LinkDemandMethod();
        }
    }
}
```

CA2142: Transparent code should not be protected with LinkDemands

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2142
Category	Microsoft.Security
Breaking change	Breaking

Cause

A transparent method requires a SecurityAction or other security demand.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule fires on transparent methods that require LinkDemands to access them. Security transparent code should not be responsible for verifying the security of an operation, and therefore should not demand permissions. Because transparent methods are supposed to be security neutral, they should not be making any security decisions. Additionally, safe critical code, which does make security decisions, should not be relying on transparent code to have previously made such a decision.

How to fix violations

To fix a violation of this rule, remove the link demand on the transparent method or mark the method with SecuritySafeCriticalAttribute attribute if it is performing security checks, such as security demands.

When to suppress warnings

Do not suppress a warning from this rule.

Example

In the following example, the rule fires on the method because the method is transparent and is marked with a LinkDemand PermissionSet that contains an SecurityAction.

```
using System;
using System.Security.Permissions;

namespace TransparencyWarningsDemo
{

    public class TransparentMethodsProtectedWithLinkDemandsClass
    {
        // CA2142 violation - transparent code using a LinkDemand.  This can be fixed by removing the
LinkDemand
        // from the method.
        [PermissionSet(SecurityAction.LinkDemand, Unrestricted = true)]
        public void TransparentMethod()
        {
        }
    }
}
```

Do not suppress a warning from this rule.

CA2143: Transparent methods should not use security demands

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2143
Category	Microsoft.Security
Breaking change	Breaking

Cause

A transparent type or method is declaratively marked with a System.Security.Permissions.SecurityAction `.Demand` demand or the method calls the System.Security.CodeAccessPermission.Demand method.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Security transparent code should not be responsible for verifying the security of an operation, and therefore should not demand permissions. Security transparent code should use full demands to make security decisions and safe-critical code should not rely on transparent code to have made the full demand. Any code that performs security checks, such as security demands, should be safe-critical instead.

How to fix violations

In general, to fix a violation of this rule, mark the method with the SecuritySafeCriticalAttribute attribute. You can also remove the demand.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The rule files on the following code because a transparent method makes a declarative security demand.

```
using System;
using System.Security;
using System.Security.Permissions;

namespace TransparencyWarningsDemo
{

    public class TransparentMethodDemandClass
    {
        // CA2142 violation - transparent code using a Demand.  This can be fixed by making the method safe
critical.
        [PermissionSet(SecurityAction.Demand, Unrestricted = true)]
        public void TransparentMethod()
        {
        }
    }
}
```

See also

CA2142: Transparent code should not be protected with LinkDemands

CA2144: Transparent code should not load assemblies from byte arrays

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2144
Category	Microsoft.Security
Breaking change	Breaking

Cause

A transparent method loads an assembly from a byte array using one of the following methods:

- Load
- Load
- Load

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

The security review for transparent code is not as thorough as the security review for critical code, because transparent code cannot perform security sensitive actions. Assemblies loaded from a byte array might not be noticed in transparent code, and that byte array might contain critical, or more importantly safe-critical code, that does need to be audited. Therefore, transparent code should not load assemblies from a byte array.

How to fix violations

To fix a violation of this rule, mark the method that is loading the assembly with the SecurityCriticalAttribute or the SecuritySafeCriticalAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The rule fires on the following code because a transparent method loads an assembly from a byte array.

```
using System;
using System.IO;
using System.Reflection;

namespace TransparencyWarningsDemo
{

    public class TransparentMethodsLoadAssembliesFromByteArraysClass
    {
        public void TransparentMethod()
        {
            byte[] assemblyBytes = File.ReadAllBytes("DependentAssembly.dll");

            // CA2144 violation - transparent code loading an assembly via byte array.  The fix here is to
            // either make TransparentMethod critical or safe-critical.
            Assembly dependent = Assembly.Load(assemblyBytes);
        }
    }
}
```

CA2145: Transparent methods should not be decorated with the SuppressUnmanagedCodeSecurityAttribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2145
Category	Microsoft.Security
Breaking change	Breaking

Cause

A transparent method, a method that is marked with the SecuritySafeCriticalAttribute method, or a type that contains a method is marked with the SuppressUnmanagedCodeSecurityAttribute attribute.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Methods decorated with the SuppressUnmanagedCodeSecurityAttribute attribute have an implicit LinkDemand placed upon any method that calls it. This LinkDemand requires that the calling code be security critical. Marking the method that uses SuppressUnmanagedCodeSecurity with the SecurityCriticalAttribute attribute makes this requirement more obvious for callers of the method.

How to fix violations

To fix a violation of this rule, mark the method or type with the SecurityCriticalAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Code

```
using System;
using System.Runtime.InteropServices;
using System.Security;

namespace TransparencyWarningsDemo
{

    public class SafeNativeMethods
    {
        // CA2145 violation - transparent method marked SuppressUnmanagedCodeSecurity.  This should be fixed
by
        // marking this method SecurityCritical.
        [DllImport("kernel32.dll", SetLastError = true)]
        [SuppressUnmanagedCodeSecurity]
        [return: MarshalAs(UnmanagedType.Bool)]
        internal static extern bool Beep(uint dwFreq, uint dwDuration);
    }
}
```

CA2146: Types must be at least as critical as their base types and interfaces

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2146
Category	Microsoft.Security
Breaking change	Breaking

Cause

A transparent type is derived from a type that is marked with the SecuritySafeCriticalAttribute or the SecurityCriticalAttribute, or a type that is marked with the SecuritySafeCriticalAttribute attribute is derived from a type that is marked with the SecurityCriticalAttribute attribute.

> **NOTE**
>
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule fires when a derived type has a security transparency attribute that is not as critical as its base type or implemented interface. Only critical types can derive from critical base types or implement critical interfaces, and only critical or safe-critical types can derive from safe-critical base types or implement safe-critical interfaces. Violations of this rule in level 2 transparency result in a TypeLoadException for the derived type.

How to fix violations

To fix this violation, mark the derived or implementing type with a transparency attribute that is at least as critical as the base type or interface.

When to suppress warnings

Do not suppress a warning from this rule.

Example

```
using System;
using System.Security;

namespace TransparencyWarningsDemo
{

    [SecuritySafeCritical]
    public class SafeCriticalBase
    {
    }

    // CA2156 violation - a transparent type cannot derive from a safe critical type.  The fix is any of:
    //   1. Make SafeCriticalBase transparent
    //   2. Make TransparentFromSafeCritical safe critical
    //   3. Make TransparentFromSafeCritical critical
    public class TransparentFromSafeCritical : SafeCriticalBase
    {
    }

    [SecurityCritical]
    public class CriticalBase
    {
    }

    // CA2156 violation - a transparent type cannot derive from a critical type.  The fix is any of:
    //   1. Make CriticalBase transparent
    //   2. Make TransparentFromCritical critical
    public class TransparentFromCritical : CriticalBase
    {
    }

    // CA2156 violation - a safe critical type cannot derive from a critical type.  The fix is any of:
    //   1. Make CriticalBase transparent
    //   2. Make CriticalBase safe critical
    //   3. Make SafeCriticalFromCritical critical
    [SecuritySafeCritical]
    public class SafeCriticalFromCritical : CriticalBase
    {
    }

}
```

CA2147: Transparent methods may not use security asserts

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2147
Category	Microsoft.Security
Breaking change	Breaking

Cause

Code that is marked as SecurityTransparentAttribute is not granted sufficient permissions to assert.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule analyzes all methods and types in an assembly that's either 100% transparent or mixed transparent/critical, and flags any declarative or imperative usage of Assert.

At run time, any calls to Assert from transparent code will cause a InvalidOperationException to be thrown. This can occur in both 100% transparent assemblies, and also in mixed transparent/critical assemblies where a method or type is declared transparent, but includes a declarative or imperative Assert.

The .NET Framework 2.0 introduced a feature named *transparency*. Individual methods, fields, interfaces, classes, and types can be either transparent or critical.

Transparent code is not allowed to elevate security privileges. Therefore, any permissions granted or demanded of it are automatically passed through the code to the caller or host application domain. Examples of elevations include Asserts, LinkDemands, SuppressUnmanagedCode, and `unsafe` code.

How to fix violations

To resolve the issue, either mark the code that calls the Assert with the SecurityCriticalAttribute, or remove the Assert.

When to suppress warnings

Do not suppress a message from this rule.

Example

This code will fail if `SecurityTestClass` is transparent, when the `Assert` method throws a InvalidOperationException.

```
using System;
using System.Security;
using System.Security.Permissions;

namespace TransparencyWarningsDemo
{

    public class TransparentMethodsUseSecurityAssertsClass
    {
        // CA2147 violation - transparent code using a security assert declaratively.  This can be fixed by
        // any of:
        //   1. Make DeclarativeAssert critical
        //   2. Make DeclarativeAssert safe critical
        //   3. Remove the assert attribute
        [PermissionSet(SecurityAction.Assert, Unrestricted = true)]
        public void DeclarativeAssert()
        {
        }

        public void ImperativeAssert()
        {
            // CA2147 violation - transparent code using a security assert imperatively.  This can be fixed
by
            // any of:
            //   1. Make ImperativeAssert critical
            //   2. Make ImperativeAssert safe critical
            //   3. Remove the assert call
            new PermissionSet(PermissionState.Unrestricted).Assert();
        }
    }
}
```

Example

One option is to code review the SecurityTransparentMethod method in the example below, and if the method is considered safe for elevation, mark SecurityTransparentMethod with secure-critical. This requires that a detailed, complete, and error-free security audit must be performed on the method together with any call-outs that occur within the method under the Assert:

```
using System;
using System.Security.Permissions;

namespace SecurityTestClassLibrary
{
    public class SecurityTestClass
    {
        [System.Security.SecurityCritical]
        void SecurityCriticalMethod()
        {
            new FileIOPermission(PermissionState.Unrestricted).Assert();

            // perform I/O operations under Assert
        }
    }
}
```

Another option is to remove the Assert from the code, and let any subsequent file I/O permission demands flow beyond SecurityTransparentMethod to the caller. This enables security checks. In this case, no security audit is needed, because the permission demands will flow to the caller and/or the application domain. Permission demands are closely controlled through security policy, hosting environment, and code-source permission grants.

See also

Security Warnings

CA2149: Transparent methods must not call into native code

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2149
Category	Microsoft.Security
Breaking change	Breaking

Cause

A method calls a native function through a method stub such as P/Invoke.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

This rule fires on any transparent method that calls directly into native code, for example, through a P/Invoke. Violations of this rule lead to a MethodAccessException in the level 2 transparency model, and a full demand for UnmanagedCode in the level 1 transparency model.

How to fix violations

To fix a violation of this rule, mark the method that calls the native code with the SecurityCriticalAttribute or SecuritySafeCriticalAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Example

```
using System;
using System.Runtime.InteropServices;

namespace TransparencyWarningsDemo
{

    public class CallNativeCodeClass
    {
        [DllImport("kernel32.dll", SetLastError = true)]
        [return: MarshalAs(UnmanagedType.Bool)]
        static extern bool Beep(uint dwFreq, uint dwDuration);

        public void CallNativeMethod()
        {
            // CA2149 violation - transparent method calling native code
            Beep(10000, 1);
        }
    }

}
```

CA2151: Fields with critical types should be security critical

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2151
Category	Microsoft.Security
Breaking change	Breaking

Cause

A security transparent field or a safe critical field is declared. Its type is specified as security critical. For example:

```
[assembly: AllowPartiallyTrustedCallers]

   [SecurityCritical]
   class Type1 { } // Security Critical type

   class Type2 // Security transparent type
   {
      Type1 m_field; // CA2151, transparent field of critical type
   }
```

In this example, `m_field` is a security transparent field of a type that is security critical.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

To use security critical types, the code that references the type must be either security critical or security safe critical. This is true even if the reference is indirect. For example, when you reference a transparent field that has a critical type, your code must be either security critical or security safe. Therefore, having a security transparent or security safe critical field is misleading because transparent code will still be unable to access the field.

How to fix violations

To fix a violation of this rule, mark the field with the SecurityCriticalAttribute attribute, or make the type that is referenced by the field either security transparent or safe critical.

```
// Fix 1: Make the referencing field security critical
[assembly: AllowPartiallyTrustedCallers]

   [SecurityCritical]
   class Type1 { } // Security Critical type

   class Type2 // Security transparent type
   {
      [SecurityCritical]
      Type1 m_field; // Fixed: critical type, critical field
   }

// Fix 2: Make the referencing field security critical
[assembly: AllowPartiallyTrustedCallers]

   class Type1 { } // Type1 is now transparent

   class Type2 // Security transparent type
   {
      [SecurityCritical]
      Type1 m_field; // Fixed: critical type, critical field
   }
```

When to suppress warnings

Do not suppress a warning from this rule.

Code

```
using System;
using System.Runtime.InteropServices;
using System.Security;

namespace TransparencyWarningsDemo
{

    public class SafeNativeMethods
    {
        // CA2145 violation - transparent method marked SuppressUnmanagedCodeSecurity.  This should be fixed
by
        // marking this method SecurityCritical.
        [DllImport("kernel32.dll", SetLastError = true)]
        [SuppressUnmanagedCodeSecurity]
        [return: MarshalAs(UnmanagedType.Bool)]
        internal static extern bool Beep(uint dwFreq, uint dwDuration);
    }
}
```

CA5122 P/Invoke declarations should not be safe critical

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5122
Category	Microsoft.Security
Breaking change	Breaking

Cause

A P/Invoke declaration has been marked with a SecuritySafeCriticalAttribute:

```
[assembly: AllowPartiallyTrustedCallers]

// ...
public class C
{
    [SecuritySafeCritical]
    [DllImport("kernel32.dll")]
    public static extern bool Beep(int frequency, int duration); // CA5122 - safe critical p/invoke
   }
```

In this example, `C.Beep(...)` has been marked as a security safe critical method.

> **NOTE**
>
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Methods are marked as SecuritySafeCritical when they perform a security sensitive operation, but are also safe to be used by transparent code. One of the fundamental rules of the security transparency model is that transparent code may never directly call native code through a P/Invoke. Therefore, marking a P/Invoke as security safe critical will not enable transparent code to call it, and is misleading for security analysis.

How to fix violations

To make a P/Invoke available to transparent code, expose a security safe critical wrapper method for it:

```
[assembly: AllowPartiallyTrustedCallers

class C
{
   [SecurityCritical]
   [DllImport("kernel32.dll", EntryPoint="Beep")]
   private static extern bool BeepPinvoke(int frequency, int duration); // Security Critical P/Invoke

   [SecuritySafeCritical]
   public static bool Beep(int frequency, int duration)
   {
      return BeepPInvoke(frequency, duration);
   }
}
```

When to suppress warnings

Do not suppress a warning from this rule.

CA2153: Avoid handling Corrupted State Exceptions

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2153
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Corrupted State Exceptions (CSEs) indicate that memory corruption exists in your process. Catching these rather than allowing the process to crash can lead to security vulnerabilities if an attacker can place an exploit into the corrupted memory region.

Rule description

CSE indicates that the state of a process has been corrupted and not caught by the system. In the corrupted state scenario, a general handler only catches the exception if you mark your method with the System.Runtime.ExceptionServices.HandleProcessCorruptedStateExceptionsAttribute attribute. By default, the Common Language Runtime (CLR) does not invoke catch handlers for CSEs.

The safest option is to allow the process to crash without catching these kinds of exceptions. Even logging code can allow attackers to exploit memory corruption bugs.

This warning triggers when catching CSEs with a general handler that catches all exceptions, for example, `catch (System.Exception e)` or `catch` with no exception parameter.

How to fix violations

To resolve this warning, do one of the following:

- Remove the HandleProcessCorruptedStateExceptionsAttribute attribute. This reverts to the default run-time behavior where CSEs are not passed to catch handlers.
- Remove the general catch handler in preference of handlers that catch specific exception types. This may include CSEs, assuming the handler code can safely handle them (rare).
- Rethrow the CSE in the catch handler, which passes the exception to the caller and should result in ending the running process.

When to suppress warnings

Do not suppress a warning from this rule.

Pseudo-code example

Violation

The following pseudo-code illustrates the pattern detected by this rule.

```
[HandleProcessCorruptedStateExceptions]
// Method that handles CSE exceptions.
void TestMethod1()
{
    try
    {
        FileStream fileStream = new FileStream("name", FileMode.Create);
    }
    catch (Exception e)
    {
        // Handle exception.
    }
}
```

Solution 1 - remove the attribute

Removing the HandleProcessCorruptedStateExceptionsAttribute attribute ensures that Corrupted State Exceptions are not handled by your method.

```
void TestMethod1()
{
    try
    {
        FileStream fileStream = new FileStream("name", FileMode.Create);
    }
    catch (Exception e)
    {
        // Handle exception.
    }
}
```

Solution 2 - catch specific exceptions

Remove the general catch handler and catch only specific exception types.

```
void TestMethod1()
{
    try
    {
        FileStream fileStream = new FileStream("name", FileMode.Create);
    }
    catch (IOException e)
    {
        // Handle IOException.
    }
    catch (UnauthorizedAccessException e)
    {
        // Handle UnauthorizedAccessException.
    }
}
```

Solution 3 - rethrow

Rethrow the exception.

```
[HandleProcessCorruptedStateExceptions]
void TestMethod1()
{
    try
    {
        FileStream fileStream = new FileStream("name", FileMode.Create);
    }
    catch (Exception e)
    {
        // Rethrow the exception.
        throw;
    }
}
```

CA2300: Do not use insecure deserializer BinaryFormatter

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2300
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Runtime.Serialization.Formatters.Binary.BinaryFormatter deserialization method was called or referenced.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Runtime.Serialization.Formatters.Binary.BinaryFormatter deserialization method calls or references. If you want to deserialize only when the Binder property is set to restrict types, disable this rule and enable rules CA2301 and CA2302 instead. Limiting which types can be deserialized can help mitigate against known remote code execution attacks, but your deserialization will still be vulnerable to denial of service attacks.

`BinaryFormatter` is insecure and can't be made secure. For more information, see the BinaryFormatter security guide.

How to fix violations

- Use a secure serializer instead, and **don't allow an attacker to specify an arbitrary type to deserialize**. For more information see the Preferred alternatives.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- This option makes code vulnerable to denial of service attacks and possible remote code execution attacks in the future. For more information, see the BinaryFormatter security guide. Restrict deserialized types. Implement a custom System.Runtime.Serialization.SerializationBinder. Before deserializing, set the `Binder` property to an instance of your custom SerializationBinder in all code paths. In the overridden BindToType method, if the type is unexpected, throw an exception to stop deserialization.

When to suppress warnings

`BinaryFormatter` is insecure and can't be made secure.

Pseudo-code examples

Violation

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

public class ExampleClass
{
    public object MyDeserialize(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        return formatter.Deserialize(new MemoryStream(bytes));
    }
}
```

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Public Class ExampleClass
    Public Function MyDeserialize(bytes As Byte()) As Object
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        Return formatter.Deserialize(New MemoryStream(bytes))
    End Function
End Class
```

Related rules

CA2301: Do not call BinaryFormatter.Deserialize without first setting BinaryFormatter.Binder

CA2302: Ensure BinaryFormatter.Binder is set before calling BinaryFormatter.Deserialize

CA2301: Do not call BinaryFormatter.Deserialize without first setting BinaryFormatter.Binder

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2301
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Runtime.Serialization.Formatters.Binary.BinaryFormatter deserialization method was called or referenced without the Binder property set.

By default, this rule analyzes the entire codebase, but this is configurable.

> **WARNING**
> Restricting types with a SerializationBinder can't prevent all attacks. For more information, see the BinaryFormatter security guide.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Runtime.Serialization.Formatters.Binary.BinaryFormatter deserialization method calls or references, when BinaryFormatter doesn't have its Binder set. If you want to disallow any deserialization with BinaryFormatter regardless of the Binder property, disable this rule and CA2302, and enable rule CA2300.

How to fix violations

- Use a secure serializer instead, and **don't allow an attacker to specify an arbitrary type to deserialize**. For more information see the Preferred alternatives.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- This option makes code vulnerable to denial of service attacks and possible remote code execution attacks in the future. For more information, see the BinaryFormatter security guide. Restrict deserialized types. Implement a custom System.Runtime.Serialization.SerializationBinder. Before deserializing, set the `Binder` property to an instance of your custom SerializationBinder in all code paths. In the overridden BindToType method, if the type is unexpected, throw an exception to stop deserialization.

When to suppress warnings

`BinaryFormatter` is insecure and can't be made secure.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2301.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2301.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2301.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2301.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2301.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

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

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) formatter.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

Solution

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

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Binder = new BookRecordSerializationBinder();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) formatter.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        formatter.Binder = New BookRecordSerializationBinder()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

Related rules

CA2300: Do not use insecure deserializer BinaryFormatter

CA2302: Ensure BinaryFormatter.Binder is set before calling BinaryFormatter.Deserialize

CA2302: Ensure BinaryFormatter.Binder is set before calling BinaryFormatter.Deserialize

8/4/2020 • 7 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2302
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Runtime.Serialization.Formatters.Binary.BinaryFormatter deserialization method was called or referenced and the Binder property may be null.

This rule is similar to CA2301, but analysis can't determine if the Binder is definitely null.

By default, this rule analyzes the entire codebase, but this is configurable.

> **WARNING**
> Restricting types with a SerializationBinder can't prevent all attacks. For more information, see the BinaryFormatter security guide.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Runtime.Serialization.Formatters.Binary.BinaryFormatter deserialization method calls or references when the Binder might be null. If you want to disallow any deserialization with BinaryFormatter regardless of the Binder property, disable this rule and CA2301, and enable rule CA2300.

How to fix violations

- Use a secure serializer instead, and **don't allow an attacker to specify an arbitrary type to deserialize**. For more information see the Preferred alternatives.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- This option makes code vulnerable to denial of service attacks and possible remote code execution attacks in the future. For more information, see the BinaryFormatter security guide. Restrict deserialized types. Implement a custom System.Runtime.Serialization.SerializationBinder. Before deserializing, set the `Binder` property to an instance of your custom SerializationBinder in all code paths. In the overridden BindToType

method, if the type is unexpected, throw an exception to stop deserialization.

When to suppress warnings

`BinaryFormatter` is insecure and can't be made secure.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2302.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2302.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2302.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2302.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2302.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

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

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class Binders
{
    public static SerializationBinder BookRecord =
        new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Binder = Binders.BookRecord;
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord)formatter.Deserialize(ms);    // CA2302 violation
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As SerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        formatter.Binder = Binders.BookRecord
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)    ' CA2302 violation
        End Using
    End Function
End Class
```

Solution

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

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class Binders
{
    public static SerializationBinder BookRecord =
        new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();

        // Ensure that Binder is always non-null before deserializing
        formatter.Binder = Binders.BookRecord ?? throw new Exception("Expected non-null binder");

        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord)formatter.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As SerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()

        ' Ensure that Binder is always non-null before deserializing
        formatter.Binder = If(Binders.BookRecord, New Exception("Expected non-null"))

        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

Violation

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

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BinaryFormatter Formatter { get; set; }

    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) this.Formatter.Deserialize(ms);    // CA2302 violation
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Property Formatter As BinaryFormatter

    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(Me.Formatter.Deserialize(ms), BookRecord)    ' CA2302 violation
        End Using
    End Function
End Class
```

Solution

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

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Binder = new BookRecordSerializationBinder();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) formatter.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        formatter.Binder = New BookRecordSerializationBinder()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

Related rules

CA2300: Do not use insecure deserializer BinaryFormatter

CA2301: Do not call BinaryFormatter.Deserialize without first setting BinaryFormatter.Binder

CA2305: Do not use insecure deserializer LosFormatter

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2305
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Web.UI.LosFormatter deserialization method was called or referenced.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Web.UI.LosFormatter deserialization method calls or references.

`LosFormatter` is insecure and can't be made secure. For more information, see the BinaryFormatter security guide.

How to fix violations

- Use a secure serializer instead, and **don't allow an attacker to specify an arbitrary type to deserialize**. For more information see the Preferred alternatives.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

`LosFormatter` is insecure and can't be made secure.

Pseudo-code examples

Violation

```
using System.IO;
using System.Web.UI;

public class ExampleClass
{
    public object MyDeserialize(byte[] bytes)
    {
        LosFormatter formatter = new LosFormatter();
        return formatter.Deserialize(new MemoryStream(bytes));
    }
}
```

```
Imports System.IO
Imports System.Web.UI

Public Class ExampleClass
    Public Function MyDeserialize(bytes As Byte()) As Object
        Dim formatter As LosFormatter = New LosFormatter()
        Return formatter.Deserialize(New MemoryStream(bytes))
    End Function
End Class
```

CA2310: Do not use insecure deserializer NetDataContractSerializer

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2310
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Runtime.Serialization.NetDataContractSerializer deserialization method was called or referenced.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Runtime.Serialization.NetDataContractSerializer deserialization method calls or references. If you want to deserialize only when the Binder property is set to restrict types, disable this rule and enable rules CA2311 and CA2312 instead. Limiting which types can be deserialized can help mitigate against known remote code execution attacks, but your deserialization will still be vulnerable to denial of service attacks.

`NetDataContractSerializer` is insecure and can't be made secure. For more information, see the BinaryFormatter security guide.

How to fix violations

- Use a secure serializer instead, and **don't allow an attacker to specify an arbitrary type to deserialize**. For more information see the Preferred alternatives.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- This option makes code vulnerable to denial of service attacks and possible remote code execution attacks in the future. For more information, see the BinaryFormatter security guide. Restrict deserialized types. Implement a custom System.Runtime.Serialization.SerializationBinder. Before deserializing, set the `Binder` property to an instance of your custom SerializationBinder in all code paths. In the overridden BindToType method, if the type is unexpected, throw an exception to stop deserialization.

When to suppress warnings

`NetDataContractSerializer` is insecure and can't be made secure.

Pseudo-code examples

Violation

```
using System.IO;
using System.Runtime.Serialization;

public class ExampleClass
{
    public object MyDeserialize(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        return serializer.Deserialize(new MemoryStream(bytes));
    }
}
```

```
Imports System.IO
Imports System.Runtime.Serialization

Public Class ExampleClass
    Public Function MyDeserialize(bytes As Byte()) As Object
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        Return serializer.Deserialize(New MemoryStream(bytes))
    End Function
End Class
```

Related rules

CA2311: Do not deserialize without first setting NetDataContractSerializer.Binder

CA2312: Ensure NetDataContractSerializer.Binder is set before deserializing

CA2311: Do not deserialize without first setting NetDataContractSerializer.Binder

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2311
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Runtime.Serialization.NetDataContractSerializer deserialization method was called or referenced without the Binder property set.

By default, this rule analyzes the entire codebase, but this is configurable.

> **WARNING**
> Restricting types with a SerializationBinder can't prevent all attacks. For more information, see the BinaryFormatter security guide.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Runtime.Serialization.NetDataContractSerializer deserialization method calls or references, when NetDataContractSerializer doesn't have its Binder set. If you want to disallow any deserialization with NetDataContractSerializer regardless of the Binder property, disable this rule and CA2312, and enable rule CA2310.

How to fix violations

- Use a secure serializer instead, and **don't allow an attacker to specify an arbitrary type to deserialize**. For more information see the Preferred alternatives.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- This option makes code vulnerable to denial of service attacks and possible remote code execution attacks in the future. For more information, see the BinaryFormatter security guide. Restrict deserialized types. Implement a custom System.Runtime.Serialization.SerializationBinder. Before deserializing, set the `Binder` property to an instance of your custom SerializationBinder in all code paths. In the overridden BindToType method, if the type is unexpected, throw an exception to stop deserialization.

When to suppress warnings

`NetDataContractSerializer` is insecure and can't be made secure.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2311.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2311.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2311.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2311.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2311.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.IO;
using System.Runtime.Serialization;

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);    // CA2311 violation
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)    ' CA2311 violation
        End Using
    End Function
End Class
```

Solution

```
using System;
using System.IO;
using System.Runtime.Serialization;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        serializer.Binder = new BookRecordSerializationBinder();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        serializer.Binder = New BookRecordSerializationBinder()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

Related rules

CA2310: Do not use insecure deserializer NetDataContractSerializer

CA2312: Ensure NetDataContractSerializer.Binder is set before deserializing

CA2312: Ensure NetDataContractSerializer.Binder is set before deserializing

8/4/2020 • 8 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2312
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Runtime.Serialization.NetDataContractSerializer deserialization method was called or referenced and the Binder property may be null.

This rule is similar to CA2311, but analysis can't determine if the Binder is definitely null.

By default, this rule analyzes the entire codebase, but this is configurable.

> **WARNING**
> Restricting types with a SerializationBinder can't prevent all attacks. For more information, see the BinaryFormatter security guide.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Runtime.Serialization.NetDataContractSerializer deserialization method calls or references when the Binder might be null. If you want to disallow any deserialization with NetDataContractSerializer regardless of the Binder property, disable this rule and CA2311, and enable rule CA2310.

`NetDataContractSerializer` is insecure and can't be made secure. For more information, see the BinaryFormatter security guide.

How to fix violations

- Use a secure serializer instead, and **don't allow an attacker to specify an arbitrary type to deserialize**. For more information see the Preferred alternatives.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- This option makes code vulnerable to denial of service attacks and possible remote code execution attacks in the future. For more information, see the BinaryFormatter security guide. Restrict deserialized types.

Implement a custom System.Runtime.Serialization.SerializationBinder. Before deserializing, set the `Binder` property to an instance of your custom SerializationBinder in all code paths. In the overridden BindToType method, if the type is unexpected, throw an exception to stop deserialization.

When to suppress warnings

`NetDataContractSerializer` is insecure and can't be made secure.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2312.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2312.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2312.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2312.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2312.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.IO;
using System.Runtime.Serialization;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class Binders
{
    public static SerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        serializer.Binder = Binders.BookRecord;
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As SerializationBinder = New BookRecordSerializationBinder()
End Class


Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        serializer.Binder = Binders.BookRecord
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)   ' CA2312 violation
        End Using
    End Function
End Class
```

Solution

```
using System;
using System.IO;
using System.Runtime.Serialization;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class Binders
{
    public static SerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();

        // Ensure that Binder is always non-null before deserializing
        serializer.Binder = Binders.BookRecord ?? throw new Exception("Expected non-null");

        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As SerializationBinder = New BookRecordSerializationBinder()
End Class


Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()

        ' Ensure that Binder is always non-null before deserializing
        serializer.Binder = If(Binders.BookRecord, New Exception("Expected non-null"))

        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

Violation

```
using System;
using System.IO;
using System.Runtime.Serialization;

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public string Author { get; set; }

    [DataMember]
    public int PageCount { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public NetDataContractSerializer Serializer { get; set; }

    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) this.Serializer.Deserialize(ms);    // CA2312 violation
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Author As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Property Serializer As NetDataContractSerializer

    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(Me.Serializer.Deserialize(ms), BookRecord)    ' CA2312 violation
        End Using
    End Function
End Class
```

Solution

```
using System;
using System.IO;
using System.Runtime.Serialization;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public string Author { get; set; }

    [DataMember]
    public int PageCount { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        serializer.Binder = new BookRecordSerializationBinder();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Author As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        serializer.Binder = New BookRecordSerializationBinder()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

Related rules

CA2310: Do not use insecure deserializer NetDataContractSerializer

CA2311: Do not deserialize without first setting NetDataContractSerializer.Binder

CA2315: Do not use insecure deserializer ObjectStateFormatter

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2315
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Web.UI.ObjectStateFormatter deserialization method was called or referenced.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Web.UI.ObjectStateFormatter deserialization method calls or references.

How to fix violations

- If possible, use a secure serializer instead, and **don't allow an attacker to specify an arbitrary type to deserialize**. Some safer serializers include:
  - System.Runtime.Serialization.DataContractSerializer
  - System.Runtime.Serialization.Json.DataContractJsonSerializer
  - System.Web.Script.Serialization.JavaScriptSerializer - Never use System.Web.Script.Serialization.SimpleTypeResolver. If you must use a type resolver, restrict deserialized types to an expected list.
  - System.Xml.Serialization.XmlSerializer
  - Newtonsoft Json.NET - Use TypeNameHandling.None. If you must use another value for TypeNameHandling, restrict deserialized types to an expected list with a custom ISerializationBinder.
  - Protocol Buffers
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.

- You've taken one of the precautions in How to fix violations.

Pseudo-code examples

Violation

```
using System.IO;
using System.Web.UI;

public class ExampleClass
{
    public object MyDeserialize(byte[] bytes)
    {
        ObjectStateFormatter formatter = new ObjectStateFormatter();
        return formatter.Deserialize(new MemoryStream(bytes));
    }
}
```

```
Imports System.IO
Imports System.Web.UI

Public Class ExampleClass
    Public Function MyDeserialize(bytes As Byte()) As Object
        Dim formatter As ObjectStateFormatter = New ObjectStateFormatter()
        Return formatter.Deserialize(New MemoryStream(bytes))
    End Function
End Class
```

CA2321: Do not deserialize with JavaScriptSerializer using a SimpleTypeResolver

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2321
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Web.Script.Serialization.JavaScriptSerializer deserialization method was called or referenced after initializing with a System.Web.Script.Serialization.SimpleTypeResolver.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Web.Script.Serialization.JavaScriptSerializer deserialization method calls or references, after initializing the JavaScriptSerializer with a System.Web.Script.Serialization.SimpleTypeResolver.

How to fix violations

- Don't initialize JavaScriptTypeResolver with a System.Web.Script.Serialization.SimpleTypeResolver.
- If your code needs to read data serialized using a SimpleTypeResolver, restrict deserialized types to an expected list by implementing a custom JavaScriptTypeResolver.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for

this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2321.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2321.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2321.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2321.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2321.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public T Deserialize<T>(string str)
    {
        JavaScriptSerializer s = new JavaScriptSerializer(new SimpleTypeResolver());
        return s.Deserialize<T>(str);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class ExampleClass
    Public Function Deserialize(Of T)(str As String) As T
        Dim s As JavaScriptSerializer = New JavaScriptSerializer(New SimpleTypeResolver())
        Return s.Deserialize(Of T)(str)
    End Function
End Class
```

Solution

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public T Deserialize<T>(string str)
    {
        JavaScriptSerializer s = new JavaScriptSerializer();
        return s.Deserialize<T>(str);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class ExampleClass
    Public Function Deserialize(Of T)(str As String) As T
        Dim s As JavaScriptSerializer = New JavaScriptSerializer()
        Return s.Deserialize(Of T)(str)
    End Function
End Class
```

Violation

```
using System.Web.Script.Serialization;

public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { get; set; }
    public AisleLocation Location { get; set; }
}

public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JavaScriptSerializer serializer = new JavaScriptSerializer(new SimpleTypeResolver());
        return serializer.Deserialize<BookRecord>(s);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class BookRecord
    Public Property Title As String
    Public Property Author As String
    Public Property Location As AisleLocation
End Class

Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(str As String) As BookRecord
        Dim serializer As JavaScriptSerializer = New JavaScriptSerializer(New SimpleTypeResolver())
        Return serializer.Deserialize(Of BookRecord)(str)
    End Function
End Class
```

Solution

```
using System;
using System.Web.Script.Serialization;

public class BookRecordTypeResolver : JavaScriptTypeResolver
{
    // For compatibility with data serialized with a JavaScriptSerializer initialized with SimpleTypeResolver.
    private static readonly SimpleTypeResolver Simple = new SimpleTypeResolver();

    public override Type ResolveType(string id)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"ResolveType('{id}')");

        if (id == typeof(BookRecord).AssemblyQualifiedName || id ==
typeof(AisleLocation).AssemblyQualifiedName)
        {
            return Simple.ResolveType(id);
        }
        else
        {
            throw new ArgumentException("Unexpected type ID", nameof(id));
        }
    }

    public override string ResolveTypeId(Type type)
    {
        return Simple.ResolveTypeId(type);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { get; set; }
    public AisleLocation Location { get; set; }
}

public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JavaScriptSerializer serializer = new JavaScriptSerializer(new BookRecordTypeResolver());
        return serializer.Deserialize<BookRecord>(s);
    }
}
```

```
Imports System
Imports System.Web.Script.Serialization

Public Class BookRecordTypeResolver
    Inherits JavaScriptTypeResolver

    ' For compatibility with data serialized with a JavaScriptSerializer initialized with SimpleTypeResolver.
    Private Dim Simple As SimpleTypeResolver = New SimpleTypeResolver()

    Public Overrides Function ResolveType(id As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        ''Console.WriteLine($"ResolveType('{id}')")

        If id = GetType(BookRecord).AssemblyQualifiedName Or id = GetType(AisleLocation).AssemblyQualifiedName
Then
            Return Simple.ResolveType(id)
        Else
            Throw New ArgumentException("Unexpected type", NameOf(id))
        End If
    End Function

    Public Overrides Function ResolveTypeId(type As Type) As String
        Return Simple.ResolveTypeId(type)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Author As String
    Public Property Location As AisleLocation
End Class

Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(str As String) As BookRecord
        Dim serializer As JavaScriptSerializer = New JavaScriptSerializer(New BookRecordTypeResolver())
        Return serializer.Deserialize(Of BookRecord)(str)
    End Function
End Class
```

Related rules

CA2322: Ensure JavaScriptSerializer is not initialized with SimpleTypeResolver before deserializing

CA2322: Ensure JavaScriptSerializer is not initialized with SimpleTypeResolver before deserializing

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2322
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A System.Web.Script.Serialization.JavaScriptSerializer deserialization method was called or referenced and the JavaScriptSerializer may have been initialized with a System.Web.Script.Serialization.SimpleTypeResolver.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds System.Web.Script.Serialization.JavaScriptSerializer deserialization method calls or references, when the JavaScriptSerializer may have been initialized with a System.Web.Script.Serialization.SimpleTypeResolver.

How to fix violations

- Ensure JavaScriptTypeResolver objects aren't initialized with a System.Web.Script.Serialization.SimpleTypeResolver.
- If your code needs to read data serialized using a SimpleTypeResolver, restrict deserialized types to an expected list by implementing a custom JavaScriptTypeResolver.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2322.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2322.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2322.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2322.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2322.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public JavaScriptSerializer Serializer { get; set; }

    public T Deserialize<T>(string str)
    {
        return this.Serializer.Deserialize<T>(str);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class ExampleClass
    Public Property Serializer As JavaScriptSerializer

    Public Function Deserialize(Of T)(str As String) As T
        Return Me.Serializer.Deserialize(Of T)(str)
    End Function
End Class
```

Solution

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public T Deserialize<T>(string str)
    {
        JavaScriptSerializer s = new JavaScriptSerializer();
        return s.Deserialize<T>(str);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class ExampleClass
    Public Function Deserialize(Of T)(str As String) As T
        Dim s As JavaScriptSerializer = New JavaScriptSerializer()
        Return s.Deserialize(Of T)(str)
    End Function
End Class
```

Violation

```
using System.Web.Script.Serialization;

public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { get; set; }
    public AisleLocation Location { get; set; }
}

public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public JavaScriptSerializer Serializer { get; set; }

    public BookRecord DeserializeBookRecord(string s)
    {
        return this.Serializer.Deserialize<BookRecord>(s);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class BookRecord
    Public Property Title As String
    Public Property Author As String
    Public Property Location As AisleLocation
End Class

Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Property Serializer As JavaScriptSerializer

    Public Function DeserializeBookRecord(str As String) As BookRecord
        Return Me.Serializer.Deserialize(Of BookRecord)(str)
    End Function
End Class
```

Solution

```
using System;
using System.Web.Script.Serialization;

public class BookRecordTypeResolver : JavaScriptTypeResolver
{
    // For compatibility with data serialized with a JavaScriptSerializer initialized with
SimpleTypeResolver.
    private static readonly SimpleTypeResolver Simple = new SimpleTypeResolver();

    public override Type ResolveType(string id)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"ResolveType('{id}')");

        if (id == typeof(BookRecord).AssemblyQualifiedName || id ==
typeof(AisleLocation).AssemblyQualifiedName)
        {
            return Simple.ResolveType(id);
        }
        else
        {
            throw new ArgumentException("Unexpected type ID", nameof(id));
        }
    }

    public override string ResolveTypeId(Type type)
    {
        return Simple.ResolveTypeId(type);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { get; set; }
    public AisleLocation Location { get; set; }
}

public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JavaScriptSerializer serializer = new JavaScriptSerializer(new BookRecordTypeResolver());
        return serializer.Deserialize<BookRecord>(s);
    }
}
```

```
Imports System
Imports System.Web.Script.Serialization

Public Class BookRecordTypeResolver
    Inherits JavaScriptTypeResolver

    ' For compatibility with data serialized with a JavaScriptSerializer initialized with SimpleTypeResolver.
    Private Dim Simple As SimpleTypeResolver = New SimpleTypeResolver()

    Public Overrides Function ResolveType(id As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        ''Console.WriteLine($"ResolveType('{id}')")

        If id = GetType(BookRecord).AssemblyQualifiedName Or id =
GetType(AisleLocation).AssemblyQualifiedName Then
            Return Simple.ResolveType(id)
        Else
            Throw New ArgumentException("Unexpected type", NameOf(id))
        End If
    End Function

    Public Overrides Function ResolveTypeId(type As Type) As String
        Return Simple.ResolveTypeId(type)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Author As String
    Public Property Location As AisleLocation
End Class

Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(str As String) As BookRecord
        Dim serializer As JavaScriptSerializer = New JavaScriptSerializer(New BookRecordTypeResolver())
        Return serializer.Deserialize(Of BookRecord)(str)
    End Function
End Class
```

Related rules

CA2321: Do not deserialize with JavaScriptSerializer using a SimpleTypeResolver

CA2326: Do not use TypeNameHandling values other than None

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2326
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

This rule fires when either of the following conditions are met:

- A Newtonsoft.Json.TypeNameHandling enumeration value, other than `None`, is referenced.
- An integer value representing a non-zero value is assigned to a TypeNameHandling variable.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds Newtonsoft.Json.TypeNameHandling values other than `None`. If you want to deserialize only when a Newtonsoft.Json.Serialization.ISerializationBinder is specified to restrict deserialized types, disable this rule and enable rules CA2327, CA2328, CA2329, and CA2330 instead.

How to fix violations

- Use TypeNameHandling's `None` value, if possible.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- Restrict deserialized types. Implement a custom Newtonsoft.Json.Serialization.ISerializationBinder. Before deserializing with Json.NET, ensure your custom ISerializationBinder is specified in the Newtonsoft.Json.JsonSerializerSettings.SerializationBinder property. In the overridden Newtonsoft.Json.Serialization.ISerializationBinder.BindToType method, if the type is unexpected, return `null` or throw an exception to stop deserialization.
  - If you restrict deserialized types, you may want to disable this rule and enable rules CA2327, CA2328, CA2329, and CA2330. Rules CA2327, CA2328, CA2329, and CA2330 help to ensure that you use a ISerializationBinder when using TypeNameHandling values other than `None`.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Pseudo-code examples

Violation

```
using Newtonsoft.Json;

public class ExampleClass
{
    public JsonSerializerSettings Settings { get; }

    public ExampleClass()
    {
        Settings = new JsonSerializerSettings();
        Settings.TypeNameHandling = TypeNameHandling.All;    // CA2326 violation.
    }
}
```

```
Imports Newtonsoft.Json

Public Class ExampleClass
    Public ReadOnly Property Settings() As JsonSerializerSettings

    Public Sub New()
        Settings = New JsonSerializerSettings()
        Settings.TypeNameHandling = TypeNameHandling.All    ' CA2326 violation.
    End Sub
End Class
```

Solution

```
using Newtonsoft.Json;

public class ExampleClass
{
    public JsonSerializerSettings Settings { get; }

    public ExampleClass()
    {
        Settings = new JsonSerializerSettings();

        // The default value of Settings.TypeNameHandling is TypeNameHandling.None.
    }
}
```

```
Imports Newtonsoft.Json

Public Class ExampleClass
    Public ReadOnly Property Settings() As JsonSerializerSettings

    Public Sub New()
        Settings = New JsonSerializerSettings()

        ' The default value of Settings.TypeNameHandling is TypeNameHandling.None.
    End Sub
End Class
```

Related rules

CA2327: Do not use insecure JsonSerializerSettings

CA2328: Ensure that JsonSerializerSettings are secure

CA2329: Do not deserialize with JsonSerializer using an insecure configuration

CA2330: Ensure that JsonSerializer has a secure configuration when deserializing

CA2327: Do not use insecure JsonSerializerSettings

8/4/2020 • 5 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2327
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

This rule fires when both of the following conditions are true for a Newtonsoft.Json.JsonSerializerSettings instance:

- The TypeNameHandling property is a value other than `None`.
- The SerializationBinder property is null.

When the JsonSerializerSettings instance is used under these circumstances:

- Initialized as a class field or property.
- As the return value of a method.
- Used as settings in a Newtonsoft.Json.JsonConvert method.
- Used as settings in a Newtonsoft.Json.JsonSerializer method.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds Newtonsoft.Json.JsonSerializerSettings instances that are configured to deserialize types specified from input, but not configured to restrict deserialized types with a Newtonsoft.Json.Serialization.ISerializationBinder. If you want to disallow deserialization of types specified from input completely, disable rules CA2327, CA2328, CA2329, and CA2330, and enable rule CA2326 instead.

How to fix violations

- Use TypeNameHandling's `None` value, if possible.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- Restrict deserialized types. Implement a custom Newtonsoft.Json.Serialization.ISerializationBinder. Before deserializing with Json.NET, ensure your custom ISerializationBinder is specified in the Newtonsoft.Json.JsonSerializerSettings.SerializationBinder property. In the overridden

Newtonsoft.Json.Serialization.ISerializationBinder.BindToType method, if the type is unexpected, return `null` or throw an exception to stop deserialization.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2327.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2327.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2327.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2327.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2327.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using Newtonsoft.Json;

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JsonSerializerSettings settings = new JsonSerializerSettings();
        settings.TypeNameHandling = TypeNameHandling.Auto;
        return JsonConvert.DeserializeObject<BookRecord>(s, settings);    // CA2327 violation
    }
}
```

```
Imports Newtonsoft.Json

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(s As String) As BookRecord
        Dim settings As JsonSerializerSettings = New JsonSerializerSettings()
        settings.TypeNameHandling = TypeNameHandling.Auto
        Return JsonConvert.DeserializeObject(Of BookRecord)(s, settings)    ' CA2327 violation
    End Function
End Class
```

Solution

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JsonSerializerSettings settings = new JsonSerializerSettings();
        settings.TypeNameHandling = TypeNameHandling.Auto;
        settings.SerializationBinder = new BookRecordSerializationBinder();
        return JsonConvert.DeserializeObject<BookRecord>(s, settings);
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(s As String) As BookRecord
        Dim settings As JsonSerializerSettings = New JsonSerializerSettings()
        settings.TypeNameHandling = TypeNameHandling.Auto
        settings.SerializationBinder = New BookRecordSerializationBinder()
        Return JsonConvert.DeserializeObject(Of BookRecord)(s, settings)
    End Function
End Class
```

Related rules

CA2326: Do not use TypeNameHandling values other than None

CA2328: Ensure that JsonSerializerSettings are secure

CA2329: Do not deserialize with JsonSerializer using an insecure configuration

CA2330: Ensure that JsonSerializer has a secure configuration when deserializing

CA2328: Ensure that JsonSerializerSettings are secure

8/4/2020 • 7 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2328
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

This rule fires when both of the following conditions might be true for a Newtonsoft.Json.JsonSerializerSettings instance:

- The TypeNameHandling property is a value other than `None`.
- The SerializationBinder property is null.

When the JsonSerializerSettings instance is used under these circumstances:

- Initialized as a class field or property.
- As the return value of a method.
- Used as settings in a Newtonsoft.Json.JsonConvert method.
- Used as settings in a Newtonsoft.Json.JsonSerializer method.

This rule is similar to CA2327, but analysis can't determine if the settings are definitely insecure.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds Newtonsoft.Json.JsonSerializerSettings instances that might be configured to deserialize types specified from input, but may not be configured to restrict deserialized types with a Newtonsoft.Json.Serialization.ISerializationBinder. If you want to disallow deserialization of types specified from input completely, disable rules CA2327, CA2328, CA2329, and CA2330, and enable rule CA2326 instead.

How to fix violations

- Use TypeNameHandling's `None` value, if possible.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

- Restrict deserialized types. Implement a custom Newtonsoft.Json.Serialization.ISerializationBinder. Before deserializing with Json.NET, ensure your custom ISerializationBinder is specified in the Newtonsoft.Json.JsonSerializerSettings.SerializationBinder property. In the overridden Newtonsoft.Json.Serialization.ISerializationBinder.BindToType method, if the type is unexpected, return `null` or throw an exception to stop deserialization.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.
- You know that the Newtonsoft.Json.JsonSerializerSettings.SerializationBinder property is always set when TypeNameHandling property is a value other than `None`.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2328.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2328.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2328.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2328.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2328.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public static class Binders
{
    public static ISerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JsonSerializerSettings settings = new JsonSerializerSettings();
        settings.TypeNameHandling = TypeNameHandling.Auto;
        settings.SerializationBinder = Binders.BookRecord;
        return JsonConvert.DeserializeObject<BookRecord>(s, settings);    // CA2328 -- settings might be
null
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As ISerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(s As String) As BookRecord
        Dim settings As JsonSerializerSettings = New JsonSerializerSettings()
        settings.TypeNameHandling = TypeNameHandling.Auto
        settings.SerializationBinder = Binders.BookRecord
        Return JsonConvert.DeserializeObject(Of BookRecord)(s, settings)   ' CA2328 -- settings might be
Nothing
    End Function
End Class
```

Solution

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public static class Binders
{
    public static ISerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JsonSerializerSettings settings = new JsonSerializerSettings();
        settings.TypeNameHandling = TypeNameHandling.Auto;

        // Ensure that SerializationBinder is non-null before deserializing
        settings.SerializationBinder = Binders.BookRecord ?? throw new Exception("Expected non-null");

        return JsonConvert.DeserializeObject<BookRecord>(s, settings);
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As ISerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(s As String) As BookRecord
        Dim settings As JsonSerializerSettings = New JsonSerializerSettings()
        settings.TypeNameHandling = TypeNameHandling.Auto

        ' Ensure that SerializationBinder is non-null before deserializing
        settings.SerializationBinder = If(Binders.BookRecord, New Exception("Expected non-null"))

        Return JsonConvert.DeserializeObject(Of BookRecord)(s, settings)
    End Function
End Class
```

Related rules

CA2326: Do not use TypeNameHandling values other than None

CA2327: Do not use insecure JsonSerializerSettings

CA2329: Do not deserialize with JsonSerializer using an insecure configuration

CA2330: Ensure that JsonSerializer has a secure configuration when deserializing

CA2329: Do not deserialize with JsonSerializer using an insecure configuration

8/4/2020 • 5 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2329
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

This rule fires when both of the following conditions are true for a Newtonsoft.Json.JsonSerializer instance:

- The TypeNameHandling property is a value other than `None`.
- The SerializationBinder property is null.

When the JsonSerializer instance is used under these circumstances:

- Initialized as a class field or property.
- To invoke a deserialization method.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds Newtonsoft.Json.JsonSerializer instances that are configured to deserialize types specified from input, but not configured to restrict deserialized types with a Newtonsoft.Json.Serialization.ISerializationBinder. If you want to disallow deserialization of types specified from input completely, disable rules CA2327, CA2328, CA2329, and CA2330, and enable rule CA2326 instead.

How to fix violations

- Use TypeNameHandling's `None` value, if possible.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- Restrict deserialized types. Implement a custom Newtonsoft.Json.Serialization.ISerializationBinder. Before deserializing with Json.NET, ensure your custom ISerializationBinder is specified in the Newtonsoft.Json.JsonSerializer.SerializationBinder property. In the overridden Newtonsoft.Json.Serialization.ISerializationBinder.BindToType method, if the type is unexpected, return `null` or throw an exception to stop deserialization.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2329.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2329.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2329.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2329.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2329.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using Newtonsoft.Json;

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(JsonReader reader)
    {
        JsonSerializer jsonSerializer = new JsonSerializer();
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto;
        return jsonSerializer.Deserialize<BookRecord>(reader);    // CA2329 violation
    }
}
```

```
Imports Newtonsoft.Json

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(reader As JsonReader) As BookRecord
        Dim jsonSerializer As JsonSerializer = New JsonSerializer()
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto
        Return JsonSerializer.Deserialize(Of BookRecord)(reader)    ' CA2329 violation
    End Function
End Class
```

Solution

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(JsonReader reader)
    {
        JsonSerializer jsonSerializer = new JsonSerializer();
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto;
        jsonSerializer.SerializationBinder = new BookRecordSerializationBinder();
        return jsonSerializer.Deserialize<BookRecord>(reader);
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(reader As JsonReader) As BookRecord
        Dim jsonSerializer As JsonSerializer = New JsonSerializer()
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto
        jsonSerializer.SerializationBinder = New BookRecordSerializationBinder()
        Return jsonSerializer.Deserialize(Of BookRecord)(reader)
    End Function
End Class
```

Related rules

CA2326: Do not use TypeNameHandling values other than None

CA2327: Do not use insecure JsonSerializerSettings

CA2328: Ensure that JsonSerializerSettings are secure

CA2330: Ensure that JsonSerializer has a secure configuration when deserializing

CA2350: Ensure DataTable.ReadXml()'s input is trusted

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2350
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The DataTable.ReadXml method was called or referenced.

Rule description

When deserializing a DataTable with untrusted input, an attacker can craft malicious input to perform a denial of service attack. There may be unknown remote code execution vulnerabilities.

For more information, see DataSet and DataTable security guidance.

How to fix violations

- If possible, use Entity Framework rather than the DataTable.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Pseudo-code examples

Violation

```
using System.Data;

public class ExampleClass
{
    public DataTable MyDeserialize(string untrustedXml)
    {
        DataTable dt = new DataTable();
        dt.ReadXml(untrustedXml);
    }
}
```

Related rules

CA2351: Ensure DataSet.ReadXml()'s input is trusted

CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks

CA2353: Unsafe DataSet or DataTable in serializable type

CA2354: Unsafe DataSet or DataTable in deserialized object graph can be vulnerable to remote code execution attack

CA2355: Unsafe DataSet or DataTable in deserialized object graph

CA2356: Unsafe DataSet or DataTable in web deserialized object graph

CA2351: Ensure DataSet.ReadXml()'s input is trusted

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2351
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The DataSet.ReadXml method was called or referenced.

Rule description

When deserializing a DataSet with untrusted input, an attacker can craft malicious input to perform a denial of service attack. There may be unknown remote code execution vulnerabilities.

For more information, see DataSet and DataTable security guidance.

How to fix violations

- If possible, use Entity Framework rather than the DataSet.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Pseudo-code examples

Violation

```
using System.Data;

public class ExampleClass
{
    public DataSet MyDeserialize(string untrustedXml)
    {
        DataSet dt = new DataSet();
        dt.ReadXml(untrustedXml);
    }
}
```

Related rules

CA2350: Ensure DataTable.ReadXml()'s input is trusted

CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks

CA2353: Unsafe DataSet or DataTable in serializable type

CA2354: Unsafe DataSet or DataTable in deserialized object graph can be vulnerable to remote code execution attack

CA2355: Unsafe DataSet or DataTable in deserialized object graph

CA2356: Unsafe DataSet or DataTable in web deserialized object graph

CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2352
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A class or struct marked with SerializableAttribute contains a DataSet or DataTable field or property, and doesn't have a GeneratedCodeAttribute.

Rule description

When deserializing untrusted input with BinaryFormatter and the deserialized object graph contains a DataSet or DataTable, an attacker can craft a malicious payload to perform a remote code execution attack.

This rule finds types which are insecure when deserialized. If your code doesn't deserialize the types found, then you don't have a deserialization vulnerability.

For more information, see DataSet and DataTable security guidance.

How to fix violations

- If possible, use Entity Framework rather than DataSet and DataTable.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- The type found by this rule is never deserialized, either directly or indirectly.
- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Pseudo-code examples

Violation

```
using System.Data;
using System.Runtime.Serialization;

[Serializable]
public class MyClass
{
    public DataSet MyDataSet { get; set; }
}
```

Related rules

CA2350: Ensure DataTable.ReadXml()'s input is trusted

CA2351: Ensure DataSet.ReadXml()'s input is trusted

CA2353: Unsafe DataSet or DataTable in serializable type

CA2354: Unsafe DataSet or DataTable in deserialized object graph can be vulnerable to remote code execution attack

CA2355: Unsafe DataSet or DataTable in deserialized object graph

CA2356: Unsafe DataSet or DataTable in web deserialized object graph

CA2353: Unsafe DataSet or DataTable in serializable type

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2353
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A class or struct marked with an XML serialization attribute or a data contract attribute contains a DataSet or DataTable field or property.

XML serialization attributes include:

- XmlAnyAttributeAttribute
- XmlAnyElementAttribute
- XmlArrayAttribute
- XmlArrayItemAttribute
- XmlChoiceIdentifierAttribute
- XmlElementAttribute
- XmlEnumAttribute
- XmlIgnoreAttribute
- XmlIncludeAttribute
- XmlRootAttribute
- XmlTextAttribute
- XmlTypeAttribute

Data contract serialization attributes include:

- DataContractAttribute
- DataMemberAttribute
- IgnoreDataMemberAttribute
- KnownTypeAttribute

Rule description

When deserializing untrusted input and the deserialized object graph contains a DataSet or DataTable, an attacker can craft a malicious payload to perform a denial of service attack. There may be unknown remote code execution vulnerabilities.

This rule finds types which are insecure when deserialized. If your code doesn't deserialize the types found, then you don't have a deserialization vulnerability.

For more information, see DataSet and DataTable security guidance.

How to fix violations

- If possible, use Entity Framework rather than DataSet and DataTable.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- The type found by this rule is never deserialized, either directly or indirectly.
- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Pseudo-code examples

Violation

```
using System.Data;
using System.Runtime.Serialization;

[XmlRoot]
public class MyClass
{
    public DataSet MyDataSet { get; set; }
}
```

Related rules

CA2350: Ensure DataTable.ReadXml()'s input is trusted

CA2351: Ensure DataSet.ReadXml()'s input is trusted

CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks

CA2354: Unsafe DataSet or DataTable in deserialized object graph can be vulnerable to remote code execution attack

CA2355: Unsafe DataSet or DataTable in deserialized object graph

CA2356: Unsafe DataSet or DataTable in web deserialized object graph

CA2354: Unsafe DataSet or DataTable in deserialized object graph can be vulnerable to remote code execution attack

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2354
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Deserializing with an System.Runtime.Serialization.IFormatter serialized, and the casted type's object graph can include a DataSet or DataTable.

This rule uses a different approach to a similar rule, CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks.

Rule description

When deserializing untrusted input with BinaryFormatter and the deserialized object graph contains a DataSet or DataTable, an attacker can craft a malicious payload to perform a remote code execution attack.

For more information, see DataSet and DataTable security guidance.

How to fix violations

- If possible, use Entity Framework rather than DataSet and DataTable.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Pseudo-code examples

Violation

```
using System.Data;
using System.IO;
using System.Runtime.Serialization;

[Serializable]
public class MyClass
{
    public MyOtherClass OtherClass { get; set; }
}

[Serializable]
public class MyOtherClass
{
    private DataSet myDataSet;
}

public class ExampleClass
{
    public MyClass Deserialize(Stream stream)
    {
        BinaryFormatter bf = new BinaryFormatter();
        return (MyClass) bf.Deserialize(stream);
    }
}
```

Related rules

CA2350: Ensure DataTable.ReadXml()'s input is trusted

CA2351: Ensure DataSet.ReadXml()'s input is trusted

CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks

CA2353: Unsafe DataSet or DataTable in serializable type

CA2355: Unsafe DataSet or DataTable in deserialized object graph

CA2356: Unsafe DataSet or DataTable in web deserialized object graph

CA2355: Unsafe DataSet or DataTable in deserialized object graph

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2355
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Deserializing when the casted or specified type's object graph can include a DataSet or DataTable.

This rule uses a different approach to a similar rule, CA2353: Unsafe DataSet or DataTable in serializable type.

The casted or specified type is evaluated when:

- Initializing a DataContractSerializer object
- Initializing a DataContractJsonSerializer object
- Initializing an XmlSerializer object
- Invoking JavaScriptSerializer.Deserialize
- Invoking JavaScriptSerializer.DeserializeObject
- Invoking XmlSerializer.FromTypes
- Invoking Newtonsoft Json.NET JsonSerializer.Deserialize
- Invoking Newtonsoft Json.NET JsonConvert.DeserializeObject

Rule description

When deserializing untrusted input with BinaryFormatter and the deserialized object graph contains a DataSet or DataTable, an attacker can craft a malicious payload to perform a denial of service attack. There may be unknown remote code execution vulnerabilities.

For more information, see DataSet and DataTable security guidance.

How to fix violations

- If possible, use Entity Framework rather than DataSet and DataTable.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over

time.

- You've taken one of the precautions in How to fix violations.

Pseudo-code examples

Violation

```
using System.Data;
using System.IO;
using System.Runtime.Serialization;

[Serializable]
public class MyClass
{
    public MyOtherClass OtherClass { get; set; }
}

[Serializable]
public class MyOtherClass
{
    private DataSet myDataSet;
}

public class ExampleClass
{
    public MyClass Deserialize(Stream stream)
    {
        BinaryFormatter bf = new BinaryFormatter();
        return (MyClass) bf.Deserialize(stream);
    }
}
```

Related rules

CA2350: Ensure DataTable.ReadXml()'s input is trusted

CA2351: Ensure DataSet.ReadXml()'s input is trusted

CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks

CA2353: Unsafe DataSet or DataTable in serializable type

CA2354: Unsafe DataSet or DataTable in deserialized object graph can be vulnerable to remote code execution attack

CA2356: Unsafe DataSet or DataTable in web deserialized object graph

CA2356: Unsafe DataSet or DataTable type in web deserialized object graph

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2356
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A method with a System.Web.Services.WebMethodAttribute or System.ServiceModel.OperationContractAttribute has a parameter that may reference a DataSet or DataTable.

This rule uses a different approach to a similar rule, CA2355: Unsafe DataSet or DataTable in deserialized object graph and will find different warnings.

Rule description

When deserializing untrusted input and the deserialized object graph contains a DataSet or DataTable, an attacker can craft a malicious payload to perform a denial of service attack. There may be unknown remote code execution vulnerabilities.

For more information, see DataSet and DataTable security guidance.

How to fix violations

- If possible, use Entity Framework rather than DataSet and DataTable.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.

Pseudo-code examples

Violation

```
using System;
using System.Data;
using System.Web.Services;

[WebService(Namespace = "http://contoso.example.com/")]
public class MyService : WebService
{
    [WebMethod]
    public string MyWebMethod(DataTable dataTable)
    {
        return null;
    }
}
```

Related rules

CA2350: Ensure DataTable.ReadXml()'s input is trusted

CA2351: Ensure DataSet.ReadXml()'s input is trusted

CA2352: Unsafe DataSet or DataTable in serializable type can be vulnerable to remote code execution attacks

CA2353: Unsafe DataSet or DataTable in serializable type

CA2354: Unsafe DataSet or DataTable in deserialized object graph can be vulnerable to remote code execution attack

CA2355: Unsafe DataSet or DataTable in deserialized object graph

CA2330: Ensure that JsonSerializer has a secure configuration when deserializing

8/4/2020 • 7 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2330
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

This rule fires when both of the following conditions might be true for a Newtonsoft.Json.JsonSerializer instance:

- The TypeNameHandling property is a value other than `None`.
- The SerializationBinder property is null.

When the JsonSerializer instance is used under these circumstances:

- Initialized as a class field or property.
- To invoke a deserialization method.

This rule is similar to CA2329, but analysis can't determine if the serializer is definitely configured insecurely.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types to inject objects with malicious side effects. An attack against an insecure deserializer could, for example, execute commands on the underlying operating system, communicate over the network, or delete files.

This rule finds Newtonsoft.Json.JsonSerializer instances that might be configured to deserialize types specified from input, but may not be configured to restrict deserialized types with a Newtonsoft.Json.Serialization.ISerializationBinder. If you want to disallow deserialization of types specified from input completely, disable rules CA2327, CA2328, CA2329, and CA2330, and enable rule CA2326 instead.

How to fix violations

- Use TypeNameHandling's `None` value, if possible.
- Make the serialized data tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- Restrict deserialized types. Implement a custom Newtonsoft.Json.Serialization.ISerializationBinder. Before deserializing with Json.NET, ensure your custom ISerializationBinder is specified in the Newtonsoft.Json.JsonSerializer.SerializationBinder property. In the overridden

Newtonsoft.Json.Serialization.ISerializationBinder.BindToType method, if the type is unexpected, return `null` or throw an exception to stop deserialization.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- You've taken one of the precautions in How to fix violations.
- You know that the Newtonsoft.Json.JsonSerializer.SerializationBinder property is always set when TypeNameHandling property is a value other than `None`.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2330.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2330.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA2330.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA2330.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA2330.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public static class Binders
{
    public static ISerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(JsonReader reader)
    {
        JsonSerializer jsonSerializer = new JsonSerializer();
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto;
        jsonSerializer.SerializationBinder = Binders.BookRecord;
        return jsonSerializer.Deserialize<BookRecord>(reader);    // CA2330 -- SerializationBinder might be
null
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As ISerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(reader As JsonReader) As BookRecord
        Dim jsonSerializer As JsonSerializer = New JsonSerializer()
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto
        jsonSerializer.SerializationBinder = Binders.BookRecord
        Return jsonSerializer.Deserialize(Of BookRecord)(reader)    ' CA2330 -- SerializationBinder might be
null
    End Function
End Class
```

Solution

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public static class Binders
{
    public static ISerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(JsonReader reader)
    {
        JsonSerializer jsonSerializer = new JsonSerializer();
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto;

        // Ensure that SerializationBinder is assigned non-null before deserializing
        jsonSerializer.SerializationBinder = Binders.BookRecord ?? throw new Exception("Expected non-null");

        return jsonSerializer.Deserialize<BookRecord>(reader);
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As ISerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(reader As JsonReader) As BookRecord
        Dim jsonSerializer As JsonSerializer = New JsonSerializer()
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto

        ' Ensure SerializationBinder is non-null before deserializing
        jsonSerializer.SerializationBinder = If(Binders.BookRecord, New Exception("Expected non-null"))

        Return jsonSerializer.Deserialize(Of BookRecord)(reader)
    End Function
End Class
```

Related rules

CA2326: Do not use TypeNameHandling values other than None

CA2327: Do not use insecure JsonSerializerSettings

CA2328: Ensure that JsonSerializerSettings are secure

CA2329: Do not deserialize with JsonSerializer using an insecure configuration

CA3001: Review code for SQL injection vulnerabilities

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3001
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches an SQL command's text.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input and SQL commands, be mindful of SQL injection attacks. An SQL injection attack can execute malicious SQL commands, compromising the security and integrity of your application. Typical techniques include using a single quotation mark or apostrophe for delimiting literal strings, two dashes for a comment, and a semicolon for the end of a statement. For more information, see SQL Injection.

This rule attempts to find input from HTTP requests reaching an SQL command's text.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that executes the SQL command, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

Use parameterized SQL commands, or stored procedures, with parameters containing the untrusted input.

When to suppress warnings

It's safe to suppress a warning from this rule if you know that the input is always validated against a known safe set of characters.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3001.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3001.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3001.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3001.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3001.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace TestNamespace
{
    public partial class WebForm : System.Web.UI.Page
    {
        public static string ConnectionString { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            string name = Request.Form["product_name"];
            using (SqlConnection connection = new SqlConnection(ConnectionString))
            {
                SqlCommand sqlCommand = new SqlCommand()
                {
                    CommandText = "SELECT ProductId FROM Products WHERE ProductName = '" + name + "'",
                    CommandType = CommandType.Text,
                };

                SqlDataReader reader = sqlCommand.ExecuteReader();
            }
        }
    }
}
```

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Linq

Namespace VulnerableWebApp
    Partial Public Class WebForm
        Inherits System.Web.UI.Page

        Public Property ConnectionString As String

        Protected Sub Page_Load(sender As Object, e As EventArgs)
            Dim name As String = Me.Request.Form("product_name")
            Using connection As SqlConnection = New SqlConnection(ConnectionString)
                Dim sqlCommand As SqlCommand = New SqlCommand With {.CommandText = "SELECT ProductId FROM
Products WHERE ProductName = '" + name + "'",
                                                                    .CommandType = CommandType.Text}
                Dim reader As SqlDataReader = sqlCommand.ExecuteReader()
            End Using
        End Sub
    End Class
End Namespace
```

Parameterized solution

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace TestNamespace
{
    public partial class WebForm : System.Web.UI.Page
    {
        public static string ConnectionString { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            string name = Request.Form["product_name"];
            using (SqlConnection connection = new SqlConnection(ConnectionString))
            {
                SqlCommand sqlCommand = new SqlCommand()
                {
                    CommandText = "SELECT ProductId FROM Products WHERE ProductName = @productName",
                    CommandType = CommandType.Text,
                };

                sqlCommand.Parameters.Add("@productName", SqlDbType.NVarChar, 128).Value = name;

                SqlDataReader reader = sqlCommand.ExecuteReader();
            }
        }
    }
}
```

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Linq

Namespace VulnerableWebApp
    Partial Public Class WebForm
        Inherits System.Web.UI.Page

        Public Property ConnectionString As String

        Protected Sub Page_Load(sender As Object, e As EventArgs)
            Dim name As String = Me.Request.Form("product_name")
            Using connection As SqlConnection = New SqlConnection(ConnectionString)
                Dim sqlCommand As SqlCommand = New SqlCommand With {.CommandText = "SELECT ProductId FROM
Products WHERE ProductName = @productName",
                                                                    .CommandType = CommandType.Text}
                sqlCommand.Parameters.Add("@productName", SqlDbType.NVarChar, 128).Value = name
                Dim reader As SqlDataReader = sqlCommand.ExecuteReader()
            End Using
        End Sub
    End Class
End Namespace
```

Stored procedure solution

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace TestNamespace
{
    public partial class WebForm : System.Web.UI.Page
    {
        public static string ConnectionString { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            string name = Request.Form["product_name"];
            using (SqlConnection connection = new SqlConnection(ConnectionString))
            {
                SqlCommand sqlCommand = new SqlCommand()
                {
                    CommandText = "sp_GetProductIdFromName",
                    CommandType = CommandType.StoredProcedure,
                };

                sqlCommand.Parameters.Add("@productName", SqlDbType.NVarChar, 128).Value = name;

                SqlDataReader reader = sqlCommand.ExecuteReader();
            }
        }
    }
}
```

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Linq

Namespace VulnerableWebApp
    Partial Public Class WebForm
        Inherits System.Web.UI.Page

        Public Property ConnectionString As String

        Protected Sub Page_Load(sender As Object, e As EventArgs)
            Dim name As String = Me.Request.Form("product_name")
            Using connection As SqlConnection = New SqlConnection(ConnectionString)
                Dim sqlCommand As SqlCommand = New SqlCommand With {.CommandText = "sp_GetProductIdFromName",
                                                                    .CommandType =
CommandType.StoredProcedure}
                sqlCommand.Parameters.Add("@productName", SqlDbType.NVarChar, 128).Value = name
                Dim reader As SqlDataReader = sqlCommand.ExecuteReader()
            End Using
        End Sub
    End Class
End Namespace
```

CA3002: Review code for XSS vulnerabilities

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3002
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches raw HTML output.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input from web requests, be mindful of cross-site scripting (XSS) attacks. An XSS attack injects untrusted input into raw HTML output, allowing the attacker to execute malicious scripts or maliciously modify content in your web page. A typical technique is putting `<script>` elements with malicious code in input. For more information, see OWASP's XSS.

This rule attempts to find input from HTTP requests reaching raw HTML output.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that outputs raw HTML, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

- Instead of outputting raw HTML, use a method or property that first HTML-encodes its input.
- HTML-encode untrusted data before outputting raw HTML.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know that the input is validated against a known safe set of characters not containing HTML.
- You know the data is HTML-encoded in a way not detected by this rule.

> **NOTE**
> This rule may report false positives for some methods or properties that HTML-encode their input.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3002.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3002.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3002.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3002.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3002.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        Response.Write("<HTML>" + input + "</HTML>");
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Me.Request.Form("in")
        Me.Response.Write("<HTML>" + input + "</HTML>")
    End Sub
End Class
```

Solution

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];

        // Example usage of System.Web.HttpServerUtility.HtmlEncode().
        Response.Write("<HTML>" + Server.HtmlEncode(input) + "</HTML>");
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Me.Request.Form("in")

        ' Example usage of System.Web.HttpServerUtility.HtmlEncode().
        Me.Response.Write("<HTML>" + Me.Server.HtmlEncode(input) + "</HTML>")
    End Sub
End Class
```

CA3003: Review code for file path injection vulnerabilities

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3003
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches the path of a file operation.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input from web requests, be mindful of using user-controlled input when specifying paths to files. An attacker may be able to read an unintended file, resulting in information disclosure of sensitive data. Or, an attacker may be able to write to an unintended file, resulting in unauthorized modification of sensitive data or compromising the server's security. A common attacker technique is Path Traversal to access files outside of the intended directory.

This rule attempts to find input from HTTP requests reaching a path in a file operation.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that writes to a file, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

- If possible, limit file paths based on user input to an explicitly known safe list. For example, if your application only needs to access "red.txt", "green.txt", or "blue.txt", only allow those values.
- Check for untrusted filenames and validate that the name is well formed.
- Use full path names when specifying paths.
- Avoid potentially dangerous constructs such as path environment variables.
- Only accept long filenames and validate long name if user submits short names.
- Restrict end user input to valid characters.

- Reject names where MAX_PATH length is exceeded.
- Handle filenames literally, without interpretation.
- Determine if the filename represents a file or a device.

When to suppress warnings

If you've validated input as described in the previous section, it's okay to suppress this warning.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3003.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3003.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3003.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3003.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3003.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.IO;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string userInput = Request.Params["UserInput"];
        // Assume the following directory structure:
        //   wwwroot\currentWebDirectory\user1.txt
        //   wwwroot\currentWebDirectory\user2.txt
        //   wwwroot\secret\allsecrets.txt
        // There is nothing wrong if the user inputs:
        //   user1.txt
        // However, if the user input is:
        //   ..\secret\allsecrets.txt
        // Then an attacker can now see all the secrets.

        // Avoid this:
        using (File.Open(userInput, FileMode.Open))
        {
            // Read a file with the name supplied by user
            // Input through request's query string and display
            // The content to the webpage.
        }
    }
}
```

```
Imports System
Imports System.IO

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim userInput As String = Me.Request.Params("UserInput")
        ' Assume the following directory structure:
        '   wwwroot\currentWebDirectory\user1.txt
        '   wwwroot\currentWebDirectory\user2.txt
        '   wwwroot\secret\allsecrets.txt
        ' There is nothing wrong if the user inputs:
        '   user1.txt
        ' However, if the user input is:
        '   ..\secret\allsecrets.txt
        ' Then an attacker can now see all the secrets.

        ' Avoid this:
        Using File.Open(userInput, FileMode.Open)
            ' Read a file with the name supplied by user
            ' Input through request's query string and display
            ' The content to the webpage.
        End Using
    End Sub
End Class
```

CA3004: Review code for information disclosure vulnerabilities

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3004
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

An exception's message, stack trace, or string representation reaches web output.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

Disclosing exception information gives attackers insight into the internals of your application, which can help attackers find other vulnerabilities to exploit.

This rule attempts to find an exception message, stack trace, or string representation being output to an HTTP response.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly catches an exception and then passes it to another assembly that outputs the exception, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

Don't output exception information to HTTP responses. Instead, provide a generic error message. See OWASP's Error Handling page for more guidance.

When to suppress warnings

If you know your web output is within your application's trust boundary and never exposed outside, it's okay to suppress this warning. This is rare. Take into consideration that your application's trust boundary and data flows may change over time.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3004.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3004.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3004.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3004.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3004.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs eventArgs)
    {
        try
        {
            object o = null;
            o.ToString();
        }
        catch (Exception e)
        {
            this.Response.Write(e.ToString());
        }
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs As EventArgs)
        Try
            Dim o As Object = Nothing
            o.ToString()
        Catch e As Exception
            Me.Response.Write(e.ToString())
        End Try
    End Sub
End Class
```

Solution

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs eventArgs)
    {
        try
        {
            object o = null;
            o.ToString();
        }
        catch (Exception e)
        {
            this.Response.Write("An error occurred. Please try again later.");
        }
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs As EventArgs)
        Try
            Dim o As Object = Nothing
            o.ToString()
        Catch e As Exception
            Me.Response.Write("An error occurred. Please try again later.")
        End Try
    End Sub
End Class
```

CA3005: Review code for LDAP injection vulnerabilities

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3005
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches an LDAP statement.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input, be mindful of Lightweight Directory Access Protocol (LDAP) injection attacks. An attacker can potentially run malicious LDAP statements against information directories. Applications that use user input to construct dynamic LDAP statements to access directory services are particularly vulnerable.

This rule attempts to find input from HTTP requests reaching an LDAP statement.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that executes an LDAP statement, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

For the user-controlled portion of LDAP statements, consider one of:

- Allow only a safe list of non-special characters.
- Disallow special character
- Escape special characters.

See OWASP's LDAP Injection Prevention Cheat Sheet for more guidance.

When to suppress warnings

If you know the input has been validated or escaped to be safe, it's okay to suppress this warning.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3005.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3005.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3005.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3005.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3005.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.DirectoryServices;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string userName = Request.Params["user"];
        string filter = "(uid=" + userName + ")";  // searching for the user entry

        // In this example, if we send the * character in the user parameter which will
        // result in the filter variable in the code to be initialized with (uid=*).
        // The resulting LDAP statement will make the server return any object that
        // contains a uid attribute.
        DirectorySearcher searcher = new DirectorySearcher(filter);
        SearchResultCollection results = searcher.FindAll();

        // Iterate through each SearchResult in the SearchResultCollection.
        foreach (SearchResult searchResult in results)
        {
            // ...
        }
    }
}
```

```
Imports System
Imports System.DirectoryServices

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(send As Object, e As EventArgs)
        Dim userName As String = Me.Request.Params(""user"")
        Dim filter As String = ""(uid="" + userName + "")""    ' searching for the user entry

        ' In this example, if we send the * character in the user parameter which will
        ' result in the filter variable in the code to be initialized with (uid=*).
        ' The resulting LDAP statement will make the server return any object that
        ' contains a uid attribute.
        Dim searcher As DirectorySearcher = new DirectorySearcher(filter)
        Dim results As SearchResultCollection = searcher.FindAll()

        ' Iterate through each SearchResult in the SearchResultCollection.
        For Each searchResult As SearchResult in results
            ' ...
        Next searchResult
    End Sub
End Class
```

CA3006: Review code for process command injection vulnerabilities

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3006
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches a process command.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input, be mindful of command injection attacks. A command injection attack can execute malicious commands on the underlying operating system, compromising the security and integrity of your server.

This rule attempts to find input from HTTP requests reaching a process command.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that starts a process, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

- If possible, avoid starting processes based on user input.
- Validate input against a known safe set of characters and length.

When to suppress warnings

If you know the input has been validated or escaped to be safe, it's safe to suppress this warning.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for

this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3006.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3006.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3006.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3006.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3006.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.Diagnostics;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        Process p = Process.Start(input);
    }
}
```

```
Imports System
Imports System.Diagnostics

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs as EventArgs)
        Dim input As String = Me.Request.Form("in")
        Dim p As Process = Process.Start(input)
    End Sub
End Class
```

CA3007: Review code for open redirect vulnerabilities

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3007
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches an HTTP response redirect.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input, be mindful of open redirect vulnerabilities. An attacker can exploit an open redirect vulnerability to use your website to give the appearance of a legitimate URL, but redirect an unsuspecting visitor to a phishing or other malicious webpage.

This rule attempts to find input from HTTP requests reaching an HTTP redirect URL.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that responds with an HTTP redirect, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

Some approaches to fixing open redirect vulnerabilities include:

- Don't allow users to initiate redirects.
- Don't allow users to specify any part of the URL in a redirect scenario.
- Restrict redirects to a predefined "allow list" of URLs.
- Validate redirect URLs.
- If applicable, consider using a disclaimer page when users are being redirected away from your site.

When to suppress warnings

If you know you've validated the input to be restricted to intended URLs, it's okay to suppress this warning.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3007.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3007.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3007.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3007.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3007.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["url"];
        this.Response.Redirect(input);
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs As EventArgs)
        Dim input As String = Me.Request.Form("url")
        Me.Response.Redirect(input)
    End Sub
End Class
```

Solution

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        if (String.IsNullOrWhiteSpace(input))
        {
            this.Response.Redirect("https://example.org/login.html");
        }
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs As EventArgs)
        Dim input As String = Me.Request.Form("in")
        If String.IsNullOrWhiteSpace(input) Then
            Me.Response.Redirect("https://example.org/login.html")
        End If
    End Sub
End Class
```

CA3008: Review code for XPath injection vulnerabilities

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3008
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches an XPath query.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input, be mindful of XPath injection attacks. Constructing XPath queries using untrusted input may allow an attacker to maliciously manipulate the query to return an unintended result, and possibly disclose the contents of the queried XML.

This rule attempts to find input from HTTP requests reaching an XPath expression.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that performs an XPath query, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

Some approaches to fixing XPath injection vulnerabilities include:

- Don't construct XPath queries from user input.
- Validate that the input only contains a safe set of characters.
- Escape quotation marks.

When to suppress warnings

If you know you've validated the input to be safe, it's okay to suppress this warning.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3008.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3008.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3008.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3008.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3008.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.Xml.XPath;

public partial class WebForm : System.Web.UI.Page
{
    public XPathNavigator AuthorizedOperations { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        string operation = Request.Form["operation"];

        // If an attacker uses this for input:
        //     ' or 'a' = 'a
        // Then the XPath query will be:
        //     authorizedOperation[@username = 'anonymous' and @operationName = '' or 'a' = 'a']
        // and it will return any authorizedOperation node.
        XPathNavigator node = AuthorizedOperations.SelectSingleNode(
            "//authorizedOperation[@username = 'anonymous' and @operationName = '" + operation + "']");
    }
}
```

```
Imports System
Imports System.Xml.XPath

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Public Property AuthorizedOperations As XPathNavigator

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim operation As String = Me.Request.Form("operation")

        ' If an attacker uses this for input:
        '     ' or 'a' = 'a
        ' Then the XPath query will be:
        '      authorizedOperation[@username = 'anonymous' and @operationName = '' or 'a' = 'a']
        ' and it will return any authorizedOperation node.
        Dim node As XPathNavigator = AuthorizedOperations.SelectSingleNode( _
            "//authorizedOperation[@username = 'anonymous' and @operationName = '" + operation + "']")
    End Sub
End Class
```

CA3009: Review code for XML injection vulnerabilities

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3009
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches raw XML output.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input, be mindful of XML injection attacks. An attacker can use XML injection to insert special characters into an XML document, making the document invalid XML. Or, an attacker could maliciously insert XML nodes of their choosing.

This rule attempts to find input from HTTP requests reaching a raw XML write.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that writes raw XML, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

Don't write raw XML. Instead, use methods or properties that XML-encode their input.

Or, XML-encode input before writing raw XML.

Or, validate user input by using sanitizers for primitive type conversion and XML encoding.

When to suppress warnings

Don't suppress warnings from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3009.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3009.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3009.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3009.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3009.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.Xml;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        XmlDocument d = new XmlDocument();
        XmlElement root = d.CreateElement("root");
        d.AppendChild(root);

        XmlElement allowedUser = d.CreateElement("allowedUser");
        root.AppendChild(allowedUser);

        allowedUser.InnerXml = "alice";

        // If an attacker uses this for input:
        //     some text<allowedUser>oscar</allowedUser>
        // Then the XML document will be:
        //     <root>some text<allowedUser>oscar</allowedUser></root>
        root.InnerXml = input;
    }
}
```

```
Imports System
Imports System.Xml

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Request.Form("in")
        Dim d As XmlDocument = New XmlDocument()
        Dim root As XmlElement = d.CreateElement("root")
        d.AppendChild(root)

        Dim allowedUser As XmlElement = d.CreateElement("allowedUser")
        root.AppendChild(allowedUser)

        allowedUser.InnerXml = "alice"

        ' If an attacker uses this for input:
        '     some text<allowedUser>oscar</allowedUser>
        ' Then the XML document will be:
        '     <root>some text<allowedUser>oscar</allowedUser></root>
        root.InnerXml = input
    End Sub
End Class
```

Solution

```
using System;
using System.Xml;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        XmlDocument d = new XmlDocument();
        XmlElement root = d.CreateElement("root");
        d.AppendChild(root);

        XmlElement allowedUser = d.CreateElement("allowedUser");
        root.AppendChild(allowedUser);

        allowedUser.InnerText = "alice";

        // If an attacker uses this for input:
        //     some text<allowedUser>oscar</allowedUser>
        // Then the XML document will be:
        //     <root>&lt;allowedUser&gt;oscar&lt;/allowedUser&gt;some text<allowedUser>alice</allowedUser>
</root>
        root.InnerText = input;
    }
}
```

```
Imports System
Imports System.Xml

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Request.Form("in")
        Dim d As XmlDocument = New XmlDocument()
        Dim root As XmlElement = d.CreateElement("root")
        d.AppendChild(root)

        Dim allowedUser As XmlElement = d.CreateElement("allowedUser")
        root.AppendChild(allowedUser)

        allowedUser.InnerText = "alice"

        ' If an attacker uses this for input:
        '     some text<allowedUser>oscar</allowedUser>
        ' Then the XML document will be:
        '     <root>&lt;allowedUser&gt;oscar&lt;/allowedUser&gt;some text<allowedUser>alice</allowedUser>
</root>
        root.InnerText = input
    End Sub
End Class
```

CA3010: Review code for XAML injection vulnerabilities

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3010
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches a System.Windows.Markup.XamlReader Load method.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input, be mindful of XAML injection attacks. XAML is a markup language that directly represents object instantiation and execution. That means elements created in XAML can interact with system resources (for example, network access and file system IO). If an attacker can control the input to a System.Windows.Markup.XamlReader Load method call, then the attacker can execute code.

This rule attempts to find input from HTTP requests that reaches a System.Windows.Markup.XamlReader Load method.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that loads XAML, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

Don't load untrusted XAML.

When to suppress warnings

Don't suppress warnings from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3010.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3010.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3010.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3010.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3010.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.IO;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        byte[] bytes = Convert.FromBase64String(input);
        MemoryStream ms = new MemoryStream(bytes);
        System.Windows.Markup.XamlReader.Load(ms);
    }
}
```

```
Imports System
Imports System.IO

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Request.Form("in")
        Dim bytes As Byte() = Convert.FromBase64String(input)
        Dim ms As MemoryStream = New MemoryStream(bytes)
        System.Windows.Markup.XamlReader.Load(ms)
    End Sub
End Class
```

CA3011: Review code for DLL injection vulnerabilities

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3011
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches a method that loads an assembly.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input, be mindful of loading untrusted code. If your web application loads untrusted code, an attacker may be able to inject malicious DLLs into your process and execute malicious code.

This rule attempts to find input from an HTTP request that reaches a method that loads an assembly.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that loads an assembly, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

Don't load untrusted DLLs from user input.

When to suppress warnings

Don't suppress warnings from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule

should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3011.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3011.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3011.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3011.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3011.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.Reflection;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        byte[] rawAssembly = Convert.FromBase64String(input);
        Assembly.Load(rawAssembly);
    }
}
```

```
Imports System
Imports System.Reflection

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Request.Form("in")
        Dim rawAssembly As Byte() = Convert.FromBase64String(input)
        Assembly.Load(rawAssembly)
    End Sub
End Class
```

CA3012: Review code for regex injection vulnerabilities

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3012
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Potentially untrusted HTTP request input reaches a regular expression.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

When working with untrusted input, be mindful of regex injection attacks. An attacker can use regex injection to maliciously modify a regular expression, to make the regex match unintended results, or to make the regex consume excessive CPU resulting in a Denial of Service attack.

This rule attempts to find input from HTTP requests reaching a regular expression.

> **NOTE**
> This rule can't track data across assemblies. For example, if one assembly reads the HTTP request input and then passes it to another assembly that creates a regular expression, this rule won't produce a warning.

> **NOTE**
> There is a configurable limit to how deep this rule will analyze data flow across method calls. See Analyzer Configuration for how to configure the limit in an EditorConfig file.

How to fix violations

Some mitigations against regex injections include:

- Always use a match timeout when using regular expressions.
- Avoid using regular expressions based on user input.
- Escape special characters from user input by calling System.Text.RegularExpressions.Regex.Escape or another method.
- Allow only non-special characters from user input.

When to suppress warnings

If you know you're using a match timeout and the user input is free of special characters, it's okay to suppress this warning.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure the analysis for this rule.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA3012.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA3012.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA3012.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA3012.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA3012.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.Text.RegularExpressions;

public partial class WebForm : System.Web.UI.Page
{
    public string SearchableText { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        string findTerm = Request.Form["findTerm"];
        Match m = Regex.Match(SearchableText, "^term=" + findTerm);
    }
}
```

```
Imports System
Imports System.Text.RegularExpressions

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Public Property SearchableText As String

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim findTerm As String = Request.Form("findTerm")
        Dim m As Match = Regex.Match(SearchableText, "^term=" + findTerm)
    End Sub
End Class
```

CA3061: Do not add schema by URL

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3061
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Overload of `XmlSchemaCollection.Add(String, String)` is using `XmlUrlResolver` to specify external XML schema in the form of an URI. If the URI String is tainted, it may lead to parsing of a malicious XML schema, which allows for the inclusion of XML bombs and malicious external entities. This could allow a malicious attacker to perform a denial of service, information disclosure, or server-side request forgery attack.

Rule description

Do not use the unsafe overload of the `Add` method because it may cause dangerous external references.

How to fix violations

- Do not use `XmlSchemaCollection.Add(String, String)`.

When to suppress warnings

Suppress this rule if you are sure your XML does not resolve dangerous external references.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule. The second parameter's type is `string`.

```
using System;
using System.Xml.Schema;
...
XmlSchemaCollection xsc = new XmlSchemaCollection();
xsc.Add("urn: bookstore - schema", "books.xsd");
```

Solution

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Schema;
...
XmlSchemaCollection xsc = new XmlSchemaCollection();
xsc.Add("urn: bookstore - schema", new XmlTextReader(new FileStream(""xmlFilename"", FileMode.Open)));
```

CA3075: Insecure DTD Processing

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3075
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

If you use insecure DtdProcessing instances or reference external entity sources, the parser may accept untrusted input and disclose sensitive information to attackers.

Rule description

A *Document Type Definition (DTD)* is one of two ways an XML parser can determine the validity of a document, as defined by the World Wide Web Consortium (W3C) Extensible Markup Language (XML) 1.0. This rule seeks properties and instances where untrusted data is accepted to warn developers about potential Information Disclosure threats or Denial of Service (DoS) attacks. This rule triggers when:

- DtdProcessing is enabled on the XmlReader instance, which resolves external XML entities using XmlUrlResolver.
- The InnerXml property in the XML is set.
- DtdProcessing property is set to Parse.
- Untrusted input is processed using XmlResolver instead of XmlSecureResolver.
- The XmlReader.Create method is invoked with an insecure XmlReaderSettings instance or no instance at all.
- XmlReader is created with insecure default settings or values.

In each of these cases, the outcome is the same: the contents from either the file system or network shares from the machine where the XML is processed will be exposed to the attacker, or DTD processing can be used as a DoS vector.

How to fix violations

- Catch and process all XmlTextReader exceptions properly to avoid path information disclosure.
- Use the XmlSecureResolver to restrict the resources that the XmlTextReader can access.
- Do not allow the XmlReader to open any external resources by setting the XmlResolver property to **null**.
- Ensure that the DataViewManager.DataViewSettingCollectionString property is assigned from a trusted source.

.NET 3.5 and earlier

- Disable DTD processing if you are dealing with untrusted sources by setting the ProhibitDtd property to **true**.
- XmlTextReader class has a full trust inheritance demand.

.NET 4 and later

- Avoid enabling DtdProcessing if you're dealing with untrusted sources by setting the XmlReaderSettings.DtdProcessing property to **Prohibit** or **Ignore**.
- Ensure that the Load() method takes an XmlReader instance in all InnerXml cases.

> **NOTE**
>
> This rule might report false positives on some valid XmlSecureResolver instances.

When to suppress warnings

Unless you're sure that the input is known to be from a trusted source, do not suppress a rule from this warning.

Pseudo-code Examples

Violation

```
using System.IO;
using System.Xml.Schema;

class TestClass
{
    public XmlSchema Test
    {
        get
        {
            var src = "";
            TextReader tr = new StreamReader(src);
            XmlSchema schema = XmlSchema.Read(tr, null); // warn
            return schema;
        }
    }
}
```

Solution

```
using System.IO;
using System.Xml;
using System.Xml.Schema;

class TestClass
{
    public XmlSchema Test
    {
        get
        {
            var src = "";
            TextReader tr = new StreamReader(src);
            XmlReader reader = XmlReader.Create(tr, new XmlReaderSettings() { XmlResolver = null });
            XmlSchema schema = XmlSchema.Read(reader , null);
            return schema;
        }
    }
}
```

Violation

```
using System.Xml;

namespace TestNamespace
{
    public class TestClass
    {
        public XmlReaderSettings settings = new XmlReaderSettings();
        public void TestMethod(string path)
        {
            var reader = XmlReader.Create(path, settings);  // warn
        }
    }
}
```

Solution

```
using System.Xml;

namespace TestNamespace
{
    public class TestClass
    {
        public XmlReaderSettings settings = new XmlReaderSettings()
        {
            DtdProcessing = DtdProcessing.Prohibit
        };

        public void TestMethod(string path)
        {
            var reader = XmlReader.Create(path, settings);
        }
    }
}
```

Violations

```
using System.Xml;

namespace TestNamespace
{
    public class DoNotUseSetInnerXml
    {
        public void TestMethod(string xml)
        {
            XmlDocument doc = new XmlDocument() { XmlResolver = null };
            doc.InnerXml = xml; // warn
        }
    }
}
```

```
using System.Xml;

namespace TestNamespace
{
    public class DoNotUseLoadXml
    {
        public void TestMethod(string xml)
        {
            XmlDocument doc = new XmlDocument(){ XmlResolver = null };
            doc.LoadXml(xml); // warn
        }
    }
}
```

Solution

```
using System.Xml;

public static void TestMethod(string xml)
{
    XmlDocument doc = new XmlDocument() { XmlResolver = null };
    System.IO.StringReader sreader = new System.IO.StringReader(xml);
    XmlReader reader = XmlReader.Create(sreader, new XmlReaderSettings() { XmlResolver = null });
    doc.Load(reader);
}
```

Violation

```
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace TestNamespace
{
    public class UseXmlReaderForDeserialize
    {
        public void TestMethod(Stream stream)
        {
            XmlSerializer serializer = new XmlSerializer(typeof(UseXmlReaderForDeserialize));
            serializer.Deserialize(stream); // warn
        }
    }
}
```

Solution

```
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace TestNamespace
{
    public class UseXmlReaderForDeserialize
    {
        public void TestMethod(Stream stream)
        {
            XmlSerializer serializer = new XmlSerializer(typeof(UseXmlReaderForDeserialize));
            XmlReader reader = XmlReader.Create(stream, new XmlReaderSettings() { XmlResolver = null });
            serializer.Deserialize(reader );
        }
    }
}
```

Violation

```
using System.Xml;
using System.Xml.XPath;

namespace TestNamespace
{
    public class UseXmlReaderForXPathDocument
    {
        public void TestMethod(string path)
        {
            XPathDocument doc = new XPathDocument(path); // warn
        }
    }
}
```

Solution

```
using System.Xml;
using System.Xml.XPath;

namespace TestNamespace
{
    public class UseXmlReaderForXPathDocument
    {
        public void TestMethod(string path)
        {
            XmlReader reader = XmlReader.Create(path, new XmlReaderSettings() { XmlResolver = null });
            XPathDocument doc = new XPathDocument(reader);
        }
    }
}
```

Violation

```
using System.Xml;

namespace TestNamespace
{
    class TestClass
    {
        public XmlDocument doc = new XmlDocument() { XmlResolver = new XmlUrlResolver() };
    }
}
```

Solution

```
using System.Xml;

namespace TestNamespace
{
    class TestClass
    {
        public XmlDocument doc = new XmlDocument() { XmlResolver = null }; // or set to a XmlSecureResolver
instance
    }
}
```

Violations

```
using System.Xml;

namespace TestNamespace
{
    class TestClass
    {
        private static void TestMethod()
        {
            var reader = XmlTextReader.Create(""doc.xml""); //warn
        }
    }
}
```

```
using System.Xml;

namespace TestNamespace
{
    public class TestClass
    {
        public void TestMethod(string path)
        {
            try {
                XmlTextReader reader = new XmlTextReader(path); // warn
            }
            catch { throw ; }
            finally {}
        }
    }
}
```

Solution

```
using System.Xml;

namespace TestNamespace
{
    public class TestClass
    {
        public void TestMethod(string path)
        {
            XmlReaderSettings settings = new XmlReaderSettings() { XmlResolver = null };
            XmlReader reader = XmlReader.Create(path, settings);
        }
    }
}
```

NOTE

Although XmlReader.Create is the recommended way to create an XmlReader instance, there are behavior differences from XmlTextReader. An XmlReader from Create normalizes `\r\n` to `\n` in XML values, while XmlTextReader preserves the `\r\n` sequence.

CA3076: Insecure XSLT Script Execution

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3076
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

If you execute Extensible Stylesheets Language Transformations (XSLT) in .NET applications insecurely, the processor may resolve untrusted URI references that could disclose sensitive information to attackers, leading to Denial of Service and Cross-Site attacks. For more information, see XSLT Security Considerations(.NET Guide).

Rule description

XSLT is a World Wide Web Consortium (W3C) standard for transforming XML data. XSLT is typically used to write style sheets to transform XML data to other formats such as HTML, fixed-length text, comma-separated text, or a different XML format. Although prohibited by default, you may choose to enable it for your project.

To ensure you're not exposing an attack surface, this rule triggers whenever the XslCompiledTransform.Load receives insecure combination instances of XsltSettings and XmlResolver, which allows malicious script processing.

How to fix violations

- Replace the insecure XsltSettings argument with XsltSettings.Default or with an instance that has disabled document function and script execution.
- Replace the XmlResolver argument with null or an XmlSecureResolver instance.

When to suppress warnings

Unless you're sure that the input is known to be from a trusted source, do not suppress a rule from this warning.

Pseudo-code Examples

Violation that uses XsltSettings.TrustedXslt

```
using System.Xml;
using System.Xml.Xsl;

namespace TestNamespace
{
    class TestClass
    {
         void TestMethod()
        {
             XslCompiledTransform xslCompiledTransform = new XslCompiledTransform();
             var settings = XsltSettings.TrustedXslt;
             var resolver = new XmlUrlResolver();
             xslCompiledTransform.Load("testStylesheet", settings, resolver); // warn
        }
    }
}
```

Solution that uses XsltSettings.Default

```
using System.Xml;
using System.Xml.Xsl;

namespace TestNamespace
{
    class TestClass
    {
        void TestMethod()
        {
            XslCompiledTransform xslCompiledTransform = new XslCompiledTransform();
            var settings = XsltSettings.Default;
            var resolver = new XmlUrlResolver();
            xslCompiledTransform.Load("testStylesheet", settings, resolver);
        }
    }
}
```

Violation—document function and script execution not disabled

```
using System.Xml;
using System.Xml.Xsl;

namespace TestNamespace
{
    class TestClass
    {
        private static void TestMethod(XsltSettings settings)
        {
            try
            {
                XslCompiledTransform xslCompiledTransform = new XslCompiledTransform();
                var resolver = new XmlUrlResolver();
                xslCompiledTransform.Load("testStylesheet", settings, resolver); // warn
            }
            catch { throw; }
            finally { }
        }
    }
}
```

Solution—disable document function and script execution

```
using System.Xml;
using System.Xml.Xsl;

namespace TestNamespace
{
    class TestClass
    {
        private static void TestMethod(XsltSettings settings)
        {
            try
            {
                XslCompiledTransform xslCompiledTransform = new XslCompiledTransform();
                settings.EnableDocumentFunction = false;
                settings.EnableScript = false;
                var resolver = new XmlUrlResolver();
                xslCompiledTransform.Load("testStylesheet", settings, resolver);
            }
            catch { throw; }
            finally { }
        }
    }
}
```

See also

- XSLT Security Considerations(.NET Guide)

CA3077: Insecure Processing in API Design, XML Document and XML Text Reader

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3077
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

When designing an API derived from XMLDocument and XMLTextReader, be mindful of DtdProcessing. Using insecure DTDProcessing instances when referencing or resolving external entity sources or setting insecure values in the XML may lead to information disclosure.

Rule description

A *Document Type Definition (DTD)* is one of two ways an XML parser can determine the validity of a document, as defined by the World Wide Web Consortium (W3C) Extensible Markup Language (XML) 1.0. This rule seeks properties and instances where untrusted data is accepted to warn developers about potential Information Disclosure threats, which may lead to Denial of Service (DoS) attacks. This rule triggers when:

- XmlDocument or XmlTextReader classes use default resolver values for DTD processing .
- No constructor is defined for the XmlDocument or XmlTextReader derived classes or no secure value is used for XmlResolver.

How to fix violations

- Catch and process all XmlTextReader exceptions properly to avoid path information disclosure .
- Use XmlSecureResolverinstead of XmlResolver to restrict the resources the XmlTextReader can access.

When to suppress warnings

Unless you're sure that the input is known to be from a trusted source, do not suppress a rule from this warning.

Pseudo-code Examples

Violation

```
using System;
using System.Xml;

namespace TestNamespace
{
    class TestClass : XmlDocument
    {
        public TestClass () {} // warn
    }

    class TestClass2 : XmlTextReader
    {
        public TestClass2() // warn
        {
        }
    }
}
```

Solution

```
using System;
using System.Xml;

namespace TestNamespace
{
    class TestClass : XmlDocument
    {
        public TestClass ()
        {
            XmlResolver = null;
        }
    }

    class TestClass2 : XmlTextReader
    {
        public TestClass2()
        {
               XmlResolver = null;
        }
    }
}
```

CA3147: Mark verb handlers with ValidateAntiForgeryToken

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA3147
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

An ASP.NET MVC controller action method isn't marked with ValidateAntiForgeryTokenAttribute, or an attribute specifying the HTTP verb, such as HttpGetAttribute or AcceptVerbsAttribute.

Rule description

When designing an ASP.NET MVC controller, be mindful of cross-site request forgery attacks. A cross-site request forgery attack can send malicious requests from an authenticated user to your ASP.NET MVC controller. For more information, see XSRF/CSRF prevention in ASP.NET MVC and web pages.

This rule checks that ASP.NET MVC controller action methods either:

- Have the ValidateAntiforgeryTokenAttribute and specify allowed HTTP verbs, not including HTTP GET.
- Specify HTTP GET as an allowed verb.

How to fix violations

- For ASP.NET MVC controller actions that handle HTTP GET requests and don't have potentially harmful side effects, add an HttpGetAttribute to the method.

  If you have an ASP.NET MVC controller action that handles HTTP GET requests and has potentially harmful side effects such as modifying sensitive data, then your application is vulnerable to cross-site request forgery attacks. You'll need to redesign your application so that only HTTP POST, PUT, or DELETE requests perform sensitive operations.

- For ASP.NET MVC controller actions that handle HTTP POST, PUT, or DELETE requests, add ValidateAntiForgeryTokenAttribute and attributes specifying the allowed HTTP verbs (AcceptVerbsAttribute, HttpPostAttribute, HttpPutAttribute, or HttpDeleteAttribute). Additionally, you need to call the HtmlHelper.AntiForgeryToken() method from your MVC view or Razor web page. For an example, see Examining the edit methods and edit view.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- The ASP.NET MVC controller action has no harmful side effects.

- The application validates the antiforgery token in a different way.

ValidateAntiForgeryToken attribute example

Violation

```
namespace TestNamespace
{
    using System.Web.Mvc;

    public class TestController : Controller
    {
        public ActionResult TransferMoney(string toAccount, string amount)
        {
            // You don't want an attacker to specify to who and how much money to transfer.

            return null;
        }
    }
}
```

Solution

```
using System;
using System.Xml;

namespace TestNamespace
{
    using System.Web.Mvc;

    public class TestController : Controller
    {
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult TransferMoney(string toAccount, string amount)
        {
            return null;
        }
    }
}
```

HttpGet attribute example

Violation

```
namespace TestNamespace
{
    using System.Web.Mvc;

    public class TestController : Controller
    {
        public ActionResult Help(int topicId)
        {
            // This Help method is an example of a read-only operation with no harmful side effects.
            return null;
        }
    }
}
```

Solution

```
namespace TestNamespace
{
    using System.Web.Mvc;

    public class TestController : Controller
    {
        [HttpGet]
        public ActionResult Help(int topicId)
        {
            return null;
        }
    }
}
```

CA5358: Do Not Use Unsafe Cipher Modes

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5358
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Use of one of the following unsafe encryption modes that is not approved:

- System.Security.Cryptography.CipherMode.ECB
- System.Security.Cryptography.CipherMode.OFB
- System.Security.Cryptography.CipherMode.CFB

Rule description

These modes are vulnerable to attacks and may cause exposure of sensitive information. For example, using `ECB` to encrypt a plaintext block always produces a same cipher text, so it can easily tell if two encrypted messages are identical. Using approved modes can avoid these unnecessary risks.

How to fix violations

- Use only approved modes (System.Security.Cryptography.CipherMode.CBC, System.Security.Cryptography.CipherMode.CTS).

When to suppress warnings

It's safe to suppress a warning from this rule if:

- Cryptography experts have reviewed and approved the cipher mode's usage.
- The referenced CipherMode isn't used for a cryptographic operation.

Pseudo-code examples

Assign ECB to Mode property

```
using System.Security.Cryptography;

class ExampleClass {
    private static void ExampleMethod () {
        RijndaelManaged rijn = new RijndaelManaged
        {
            Mode = CipherMode.ECB
        };
    }
}
```

Using the value ECB

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    private static void ExampleMethod()
    {
        Console.WriteLine(CipherMode.ECB);
    }
}
```

Solution

```
using System.Security.Cryptography;

class ExampleClass {
    private static void ExampleMethod () {
        RijndaelManaged rijn = new RijndaelManaged
        {
            Mode = CipherMode.CBC
        };
    }
}
```

CA5359: Do not disable certificate validation

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5359
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The callback assigned to ServicePointManager.ServerCertificateValidationCallback always returns `true`.

Rule description

A certificate can help authenticate the identity of the server. Clients should validate the server certificate to ensure requests are sent to the intended server. If the ServicePointManager.ServerCertificateValidationCallback always returns `true`, then by default any certificate will pass validation for all outgoing HTTPS requests.

How to fix violations

- Considering overriding certificate validation logic on the specific outgoing HTTPS requests that require custom certificate validation, instead of overriding the global ServicePointManager.ServerCertificateValidationCallback.
- Apply custom validation logic to only specific hostnames and certificates, and otherwise check that the SslPolicyErrors enum value is `None`.

When to suppress warnings

If multiple delegates are attached to ServerCertificateValidationCallback, only the value from the last delegate is respected, so it's safe to suppress warnings from other delegates. However, you may want to remove the unused delegates entirely.

Pseudo-code examples

Violation

```
using System.Net;

class ExampleClass
{
    public void ExampleMethod()
    {
        ServicePointManager.ServerCertificateValidationCallback += (sender, cert, chain, error) => { return
true; };
    }
}
```

Solution

```
using System.Net;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;

class ExampleClass
{
    public void ExampleMethod()
    {
        ServicePointManager.ServerCertificateValidationCallback += SelfSignedForLocalhost;
    }

    private static bool SelfSignedForLocalhost(object sender, X509Certificate certificate, X509Chain chain,
SslPolicyErrors sslPolicyErrors)
    {
        if (sslPolicyErrors == SslPolicyErrors.None)
        {
            return true;
        }

        // For HTTPS requests to this specific host, we expect this specific certificate.
        // In practice, you'd want this to be configurable and allow for multiple certificates per host, to
enable
        // seamless certificate rotations.
        return sender is HttpWebRequest httpWebRequest
                && httpWebRequest.RequestUri.Host == "localhost"
                && certificate is X509Certificate2 x509Certificate2
                && x509Certificate2.Thumbprint == "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
                && sslPolicyErrors == SslPolicyErrors.RemoteCertificateChainErrors;
    }
}
```

CA5360: Do not call dangerous methods in deserialization

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5360
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Calling one of the following dangerous methods in deserialization:

- System.IO.Directory.Delete
- System.IO.DirectoryInfo.Delete
- System.IO.File.AppendAllLines
- System.IO.File.AppendAllText
- System.IO.File.AppendText
- System.IO.File.Copy
- System.IO.File.Delete
- System.IO.File.WriteAllBytes
- System.IO.File.WriteAllLines
- System.IO.File.WriteAllText
- System.IO.FileInfo.Delete
- System.IO.Log.LogStore.Delete
- System.Reflection.Assembly.GetLoadedModules
- System.Reflection.Assembly.Load
- System.Reflection.Assembly.LoadFrom
- System.Reflection.Assembly.LoadFile
- System.Reflection.Assembly.LoadModule
- System.Reflection.Assembly.LoadWithPartialName
- System.Reflection.Assembly.ReflectionOnlyLoad
- System.Reflection.Assembly.ReflectionOnlyLoadFrom
- System.Reflection.Assembly.UnsafeLoadFrom

All methods meets one of the following requirements could be the callback of deserialization:

- Marked with System.Runtime.Serialization.OnDeserializingAttribute.
- Marked with System.Runtime.Serialization.OnDeserializedAttribute.
- Implementing System.Runtime.Serialization.IDeserializationCallback.OnDeserialization.
- Implementing System.IDisposable.Dispose.
- Is a destructor.

Rule description

Insecure deserialization is a vulnerability which occurs when untrusted data is used to abuse the logic of an application, inflict a Denial-of-Service (DoS) attack, or even execute arbitrary code upon it being deserialized. It's frequently possible for malicious users to abuse these deserialization features when the application is deserializing untrusted data which is under their control. Specifically, invoke dangerous methods in the process of deserialization. Successful insecure deserialization attacks could allow an attacker to carry out attacks such as DoS attacks, authentication bypasses, and remote code execution.

How to fix violations

Remove these dangerous methods from automatically run deserialization callbacks. Call dangerous methods only after validating the input.

When to suppress warnings

It's safe to suppress this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- The serialized data is tamper-proof. After serialization, cryptographically sign the serialized data. Before deserialization, validate the cryptographic signature. Protect the cryptographic key from being disclosed and design for key rotations.
- The data is validated as safe to the application.

Pseudo-code examples

Violation

```
using System;
using System.IO;
using System.Runtime.Serialization;

[Serializable()]
public class ExampleClass : IDeserializationCallback
{
    private string member;

    void IDeserializationCallback.OnDeserialization(Object sender)
    {
        var sourceFileName = "malicious file";
        var destFileName = "sensitive file";
        File.Copy(sourceFileName, destFileName);
    }
}
```

Solution

```
using System;
using System.IO;
using System.Runtime.Serialization;

[Serializable()]
public class ExampleClass : IDeserializationCallback
{
    private string member;

    void IDeserializationCallback.OnDeserialization(Object sender)
    {
        var sourceFileName = "malicious file";
        var destFileName = "sensitive file";
        // Remove the potential dangerous operation.
        // File.Copy(sourceFileName, destFileName);
    }
}
```

CA5361: Do not disable SChannel use of strong crypto

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5361
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A AppContext.SetSwitch method call sets `Switch.System.Net.DontEnableSchUseStrongCrypto` to `true`.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

Setting `Switch.System.Net.DontEnableSchUseStrongCrypto` to `true` weakens the cryptography used in outgoing Transport Layer Security (TLS) connections. Weaker cryptography can compromise the confidentiality of communication between your application and the server, making it easier for attackers to eavesdrop sensitive data. For more information, see Transport Layer Security (TLS) best practices with the .NET Framework.

How to fix violations

- If your application targets .NET Framework v4.6 or later, you can either remove the AppContext.SetSwitch method call, or set the switch's value to `false`.
- If your application targets .NET Framework earlier than v4.6 and runs on .NET Framework v4.6 or later, set the switch's value to `false`.
- Otherwise, refer to Transport Layer Security (TLS) best practices with the .NET Framework for mitigations.

When to suppress warnings

You can suppress this warning if you need to connect to a legacy service that can't be upgraded to use secure TLS configurations.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5361.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5361.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5361.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5361.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5361.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5361 violation
        AppContext.SetSwitch("Switch.System.Net.DontEnableSchUseStrongCrypto", true);
    }
}
```

```
Imports System

Public Class ExampleClass
    Public Sub ExampleMethod()
        ' CA5361 violation
        AppContext.SetSwitch("Switch.System.Net.DontEnableSchUseStrongCrypto", true)
    End Sub
End Class
```

Solution

```
using System;

public class ExampleClass
{
    public void ExampleMethod()
    {
        AppContext.SetSwitch("Switch.System.Net.DontEnableSchUseStrongCrypto", false);
    }
}
```

```
Imports System

Public Class ExampleClass
    Public Sub ExampleMethod()
        AppContext.SetSwitch("Switch.System.Net.DontEnableSchUseStrongCrypto", false)
    End Sub
End Class
```

CA5362: Potential reference cycle in deserialized object graph

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5362
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A class marked with the System.SerializableAttribute has a field or property may refer to the containing object directly or indirectly, allowing for a potential reference cycle.

Rule description

If deserializing untrusted data, then any code processing the deserialized object graph needs to handle reference cycles without going into infinite loops. This includes both code that's part of a deserialization callback and code that processes the object graph after deserialization completed. Otherwise, an attacker could perform a Denial-of-Service attack with malicious data containing a reference cycle.

This rule doesn't necessarily mean there's a vulnerability, but just flags potential reference cycles in deserialized object graphs.

How to fix violations

Don't serialize the class and remove the SerializableAttribute. Or, redesign your application so that the self-referred members can be removed out of the serializable class.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- You know the input is trusted. Consider that your application's trust boundary and data flows may change over time.
- All code processing the deserialized data detects and handles reference cycles without going into an infinite loop or using excessive resources.

Pseudo-code examples

Potential reference cycle violation

```
using System;

[Serializable()]
class ExampleClass
{
    public ExampleClass ExampleProperty {get; set;}

    public int NormalProperty {get; set;}
}

class AnotherClass
{
    // The argument passed by could be `JsonConvert.DeserializeObject<ExampleClass>(untrustedData)`.
    public void AnotherMethod(ExampleClass ec)
    {
        while(ec != null)
        {
            Console.WriteLine(ec.ToString());
            ec = ec.ExampleProperty;
        }
    }
}
```

Solution

```
using System;

[Serializable()]
class ExampleClass
{
    [NonSerialized]
    public ExampleClass ExampleProperty {get; set;}

    public int NormalProperty {get; set;}
}

class AnotherClass
{
    // The argument passed by could be `JsonConvert.DeserializeObject<ExampleClass>(untrustedData)`.
    public void AnotherMethod(ExampleClass ec)
    {
        while(ec != null)
        {
            Console.WriteLine(ec.ToString());
            ec = ec.ExampleProperty;
        }
    }
}
```

CA5363: Do not disable request validation

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5363
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The attribute `ValidateInput` is set to `false` for a class or method.

Rule description

Request validation is a feature in ASP.NET that examines HTTP requests and determines whether they contain potentially dangerous content that can lead to injection attacks, including cross-site-scripting.

How to fix violations

Set the `ValidateInput` attribute to `true` or delete it entirely. Alternatively, use `AllowHTMLAttribute` to allow HTML in specific parts of the input.

When to suppress warnings

You can suppress this violation if all the payload in the incoming HTTP request is sourced from a trusted entity and could not be tampered with by an adversary prior to or during transport.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule. This disables input validation.

```
using System.Web.Mvc;

class TestControllerClass
{
    [ValidateInput(false)]
    public void TestActionMethod()
    {
    }
}
```

Solution

```
using System.Web.Mvc;

class TestControllerClass
{
    [ValidateInput(true)]
    public void TestActionMethod()
    {
    }
}
```

CA5364: Do not use deprecated security protocols

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5364
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

This rule fires when either of the following conditions are met:

- A deprecated System.Net.SecurityProtocolType value was referenced.
- An integer value representing a deprecated value was assigned to a SecurityProtocolType variable.

Deprecated values are:

- Ssl3
- Tls
- Tls10
- Tls11

Rule description

Transport Layer Security (TLS) secures communication between computers, most commonly with Hypertext Transfer Protocol Secure (HTTPS). Older protocol versions of TLS are less secure than TLS 1.2 and TLS 1.3 and are more likely to have new vulnerabilities. Avoid older protocol versions to minimize risk. For guidance on identifying and removing deprecated protocol versions, see Solving the TLS 1.0 Problem, 2nd Edition.

How to fix violations

Don't use deprecated TLS protocol versions.

When to suppress warnings

You can suppress this warning if:

- The reference to the deprecated protocol version isn't being used to enable a deprecated version.
- You need to connect to a legacy service that can't be upgraded to use secure TLS configurations.

Pseudo-code examples

Enumeration name violation

```
using System;
using System.Net;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5364 violation
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls11 | SecurityProtocolType.Tls12;
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5364 violation
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls11 Or SecurityProtocolType.Tls12
    End Sub
End Class
```

Integer value violation

```
using System;
using System.Net;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5364 violation
        ServicePointManager.SecurityProtocol = (SecurityProtocolType) 768;    // TLS 1.1
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5364 violation
        ServicePointManager.SecurityProtocol = CType(768, SecurityProtocolType)   ' TLS 1.1
    End Sub
End Class
```

Solution

```
using System;
using System.Net;

public class TestClass
{
    public void TestMethod()
    {
        // Let the operating system decide what TLS protocol version to use.
        // See https://docs.microsoft.com/dotnet/framework/network-programming/tls
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' Let the operating system decide what TLS protocol version to use.
        ' See https://docs.microsoft.com/dotnet/framework/network-programming/tls
    End Sub
End Class
```

Related rules

CA5386: Avoid hardcoding SecurityProtocolType value

CA5397: Do not use deprecated SslProtocols values

CA5398: Avoid hardcoded SslProtocols values

CA5365: Do Not Disable HTTP Header Checking

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5365
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Set EnableHeaderChecking to `false`.

Rule description

HTTP header checking enables encoding of the carriage return and newline characters, `\r` and `\n`, that are found in response headers. This encoding can help to avoid injection attacks that exploit an application that echoes untrusted data contained in the header.

How to fix violations

Set EnableHeaderChecking to `true`. Or, remove the assignment to `false` because the default value is `true`.

When to suppress warnings

HTTP header continuations rely on headers spanning multiple lines and require new lines in them. If you need to use header continuations, you need to set the EnableHeaderChecking property to `false`. There is a performance impact from checking the headers. If you are certain you are already doing the right checks, turning off this feature can improve the performance of your application. Before you disable this feature, be sure you are already taking the right precautions in this area.

Pseudo-code examples

```
using System;
using System.Web.Configuration;

class TestClass
{
    public void TestMethod()
    {
        HttpRuntimeSection httpRuntimeSection = new HttpRuntimeSection();
        httpRuntimeSection.EnableHeaderChecking = false;
    }
}
```

Solution

```
using System;
using System.Web.Configuration;

class TestClass
{
    public void TestMethod()
    {
        HttpRuntimeSection httpRuntimeSection = new HttpRuntimeSection();
        httpRuntimeSection.EnableHeaderChecking = true;
    }
}
```

CA5366: Use XmlReader For DataSet Read XML

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5366
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A Document Type Definition (DTD) defines the structure and the legal elements and attributes of an XML document. Referring to a DTD from an external resource could cause potential Denial of Service (DoS) attacks. Most readers cannot disable DTD processing and restrict external references loading except for System.Xml.XmlReader. Using these other readers to load XML by one of the following methods triggers this rule:

- ReadXml
- ReadXmlSchema
- ReadXmlSerializable

Rule description

Using a System.Data.DataSet to read XML with untrusted data may load dangerous external references, which should be restricted by using an XmlReader with a secure resolver or with DTD processing disabled.

How to fix violations

Use XmlReader or its derived classes to read XML.

When to suppress warnings

Suppress a warning from this rule when dealing with a trusted data source.

Pseudo-code examples

Violation

```
using System.Data;
using System.IO;

public class ExampleClass
{
    public void ExampleMethod()
    {
        new DataSet().ReadXml(new FileStream("xmlFilename", FileMode.Open));
    }
}
```

Solution

```
using System.Data;
using System.IO;
using System.Xml;

public class ExampleClass
{
    public void ExampleMethod()
    {
        new DataSet().ReadXml(new XmlTextReader(new FileStream("xmlFilename", FileMode.Open)));
    }
}
```

CA5367: Do not serialize types with pointer fields

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5367
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Pointers are not type safe, which means you cannot guarantee the correctness of the memory they point at. So, serializing types with pointer fields is a security risk, as it may allow an attacker to control the pointer.

Rule description

This rule checks whether there's a serializable class with a pointer field or property. Members that can't be serialized can be a pointer, such as static members or fields marked with System.NonSerializedAttribute.

How to fix violations

Don't use pointer types for members in a serializable class or don't serialize the members that are pointers.

When to suppress warnings

Don't take the risk to use pointers in serializable types.

Pseudo-code examples

Violation

```
using System;

[Serializable()]
unsafe class TestClassA
{
    private int* pointer;
}
```

Solution 1

```
using System;

[Serializable()]
unsafe class TestClassA
{
    private int i;
}
```

Solution 2

```
using System;

[Serializable()]
unsafe class TestClassA
{
    private static int* pointer;
}
```

CA5368: Set ViewStateUserKey For Classes Derived From Page

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5368
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The Page.ViewStateUserKey property is not assigned in Page.OnInit or the `Page_Init` method.

Rule description

When designing an ASP.NET Web Form, be mindful of cross-site request forgery (CSRF) attacks. A CSRF attack can send malicious requests from an authenticated user to your ASP.NET Web Form.

One way of protecting against CSRF attacks in ASP.NET Web Form is by setting a page's ViewStateUserKey to a string that is unpredictable and unique to a session. For more information, see Take Advantage of ASP.NET Built-in Features to Fend Off Web Attacks.

How to fix violations

Set the ViewStateUserKey property to a unpredictable and unique string per session. For example, if you use ASP.NET session state, HttpSessionState.SessionID will work.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- The ASP.NET Web Form page does not perform sensitive operations.
- Cross-site request forgery attacks are mitigated in a way that this rule doesn't detect. For example, if the page inherits from a master page that contains CSRF defenses.

Pseudo-code examples

Violation

```
using System;
using System.Web.UI;

class ExampleClass : Page
{
    protected override void OnInit (EventArgs e)
    {
    }
}
```

Solution

```
using System;
using System.Web.UI;

class ExampleClass : Page
{
    protected override void OnInit (EventArgs e)
    {
        // Assuming that your page makes use of ASP.NET session state and the SessionID is stable.
        ViewStateUserKey = Session.SessionID;
    }
}
```

CA5369: Use XmlReader for Deserialize

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5369
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Deserializing untrusted XML input with XmlSerializer.Deserialize instantiated without an `XmlReader` object can potentially lead to denial of service, information disclosure, and server-side request forgery attacks. These attacks are enabled by untrusted DTD and XML schema processing, which allows for the inclusion of XML bombs and malicious external entities in the XML. Only with `XmlReader` is it possible to disable DTD. Inline XML schema processing as `XmlReader` has the `ProhibitDtd` and `ProcessInlineSchema` property set to `false` by default in .NET Framework version 4.0 and later. The other options such as `Stream`, `TextReader`, and `XmlSerializationReader` cannot disable DTD processing.

Rule description

Processing untrusted DTD and XML schemas may enable loading dangerous external references, which should be restricted by using an `XmlReader` with a secure resolver or with DTD and XML inline schema processing disabled. This rule detects code that uses the XmlSerializer.Deserialize method and does not take `XmlReader` as a constructor parameter.

How to fix violations

Do not use XmlSerializer.Deserialize overloads other than Deserialize(XmlReader), Deserialize(XmlReader, String), Deserialize(XmlReader, XmlDeserializationEvents), or Deserialize(XmlReader, String, XmlDeserializationEvents).

When to suppress warnings

You can potentially suppress this warning if the parsed XML comes from a trusted source and hence cannot be tampered with.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule. The type of the first parameter of XmlSerializer.Deserialize is not `XmlReader` or a derived class thereof.

```
using System.IO;
using System.Xml.Serialization;
...
new XmlSerializer(typeof(TestClass).Deserialize(new FileStream("filename", FileMode.Open));
```

Solution

```
using System.IO;
using System.Xml;
using System.Xml.Serialization;
...
new XmlSerializer(typeof(TestClass)).Deserialize(XmlReader.Create (new FileStream("filename",
FileMode.Open)));
```

CA5370: Use XmlReader for validating reader

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5370
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Validating untrusted XML input with the `XmlValidatingReader` class instantiated without an `XmlReader` object can potentially lead to denial of service, information disclosure, and server-side request forgery. These attacks are enabled by untrusted DTD and XML schema processing, which allows for the inclusion of XML bombs and malicious external entities in the XML. Only with `XmlReader` is it possible to disable DTD. Inline XML schema processing as `XmlReader` has the `ProhibitDtd` and `ProcessInlineSchema` property set to `false` by default in .NET Framework starting in version 4.0.

Rule description

Processing untrusted DTD and XML schemas may enable loading dangerous external references. This dangerous loading can be restricted by using an `XmlReader` with a secure resolver or with DTD and XML inline schema processing disabled. This rule detects code that uses the `XmlValidatingReader` class without `XmlReader` as a constructor parameter.

How to fix violations

- Use `XmlValidatingReader(XmlReader)` with `ProhibitDtd` and `ProcessInlineSchema` properties set to `false`.
- Starting in .NET Framework 2.0, `XmlValidatingReader` is considered obsolete. You can instantiate a validating reader with XmlReader.Create.

When to suppress warnings

You can potentially suppress this warning if the `XmlValidatingReader` is always used to validate XML that comes from a trusted source and hence cannot be tampered with.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule. The type of the first parameter of `XmlValidatingReader.XmlValidatingReader()` is not `XmlReader`.

```
using System;
using System.IO;
using System.Xml;
...
public void TestMethod(Stream xmlFragment, XmlNodeType fragType, XmlParserContext context)
{
    var obj = new XmlValidatingReader(xmlFragment, fragType, context);
}
```

Solution

```
using System;
using System.Xml;
...
public void TestMethod(XmlReader xmlReader)
{
    var obj = new XmlValidatingReader(xmlReader);
}
```

CA5371: Use XmlReader for schema read

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5371
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Processing untrusted XML input with XmlSchema.Read instantiated without an `XmlReader` object can potentially lead to denial of service, information disclosure, and server-side request forgery attacks. These attacks are enabled by untrusted DTD and XML schema processing, which allows for the inclusion of XML bombs and malicious external entities in the XML. Only with `XmlReader` is it possible to disable DTD. Inline XML schema processing as `XmlReader` has the `ProhibitDtd` and `ProcessInlineSchema` property set to false by default in .NET Framework starting in version 4.0. The other options such as `Stream`, `TextReader`, and `XmlSerializationReader` cannot disable DTD processing.

Rule description

Processing untrusted DTD and XML schemas may enable loading dangerous external references. Using an `XmlReader` with a secure resolver or with DTD and XML inline schema processing disabled restricts this. This rule detects code that uses the XmlSchema.Read method without `XmlReader` as a parameter.

How to fix violations

Use `XmlSchema.Read(XmlReader, *)` overloads.

When to suppress warnings

You can potentially suppress this warning if the XmlSchema.Read method is always used to process XML that comes from a trusted source and hence cannot be tampered with.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule. The type of the first parameter of XmlSchema.Read is not `XmlReader`.

```
using System.IO;
using System.Xml.Schema;
...
public void TestMethod(Stream stream, ValidationEventHandler validationEventHandler)
{
    XmlSchema.Read(stream, validationEventHandler);
}
```

Solution

```
using System.IO;
using System.Xml.Schema;
...
public void TestMethod(XmlReader reader, ValidationEventHandler validationEventHandler)
{
    XmlSchema.Read(reader, validationEventHandler);
}
```

CA5372: Use XmlReader for XPathDocument

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5372
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using the `XPathDocument` class instantiated without an `XmlReader` object can potentially lead to denial of service, information disclosure, and server-side request forgery attacks. These attacks are enabled by untrusted DTD and XML schema processing, which allows for the inclusion of XML bombs and malicious external entities in the XML. Only with `XmlReader` is it possible to disable DTD. Inline XML schema processing as `XmlReader` has the `ProhibitDtd` and `ProcessInlineSchema` property set to false by default in .NET Framework starting in version 4.0. The other options such as `Stream`, `TextReader`, and `XmlSerializationReader` cannot disable DTD processing.

Rule description

Processing XML from untrusted data may load dangerous external references, which can be restricted by using an `XmlReader` with a secure resolver or with DTD processing disabled. This rule detects code that uses the `XPathDocument` class and doesn't take `XmlReader` as a constructor parameter.

How to fix violations

Use `XPathDocument(XmlReader, *)` constructors.

When to suppress warnings

You can potentially suppress this warning if the `XPathDocument` object is used to process an XML file that comes from a trusted source and hence cannot be tampered with.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule. The type of the first parameter of `XPathDocument` is not `XmlReader`.

```
using System.IO;
using System.Xml.XPath;
...
var obj = new XPathDocument(stream);
```

Solution

```
using System.Xml;
using System.Xml.XPath;
...
public void TestMethod(XmlReader reader)
{
var obj = new XPathDocument(reader);
}
```

CA5373: Do not use obsolete key derivation function

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5373
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Cryptographically weak key derivation methods System.Security.Cryptography.PasswordDeriveBytes and/or Rfc2898DeriveBytes.CryptDeriveKey are used to generate a key.

Rule description

This rule detects the invocation of weak key derivation methods System.Security.Cryptography.PasswordDeriveBytes and Rfc2898DeriveBytes.CryptDeriveKey. System.Security.Cryptography.PasswordDeriveBytes used a weak algorithm PBKDF1. Rfc2898DeriveBytes.CryptDeriveKey does not use iteration count and salt from the `Rfc2898DeriveBytes` object, which makes it weak.

How to fix violations

Password-based key derivation should use the PBKDF2 algorithm with SHA-2 hashing. Rfc2898DeriveBytes.GetBytes can be used to achieve that.

When to suppress warnings

Suppress the warning if the risk associated with using PBKDF1 is carefully reviewed and accepted.

Pseudo-code examples

Violation

As of the time of this writing, the following pseudo-code sample illustrates the pattern detected by this rule.

```
using System;
using System.Security.Cryptography;
class TestClass
{
    public void TestMethod(Rfc2898DeriveBytes rfc2898DeriveBytes, string algname, string alghashname, int
keySize, byte[] rgbIV)
    {
        rfc2898DeriveBytes.CryptDeriveKey(algname, alghashname, keySize, rgbIV);
    }
}
```

Solution

```
using System;
using System.Security.Cryptography;
class TestClass
{
    public void TestMethod(Rfc2898DeriveBytes rfc2898DeriveBytes)
    {
        rfc2898DeriveBytes.GetBytes(1);
    }
}
```

CA5374: Do not use XslTransform

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5374
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Instantiating an System.Xml.Xsl.XslTransform, which doesn't restrict potentially dangerous external references or prevent scripts.

Rule description

XslTransform is vulnerable when operating on untrusted input. An attack could execute arbitrary code.

How to fix violations

Replace XslTransform with System.Xml.Xsl.XslCompiledTransform. For more guidance, see [/dotnet/standard/data/xml/migrating-from-the-xsltransform-class].

When to suppress warnings

The XslTransform object, XSLT style sheets, and XML source data are all from trusted sources.

Pseudo-code examples

Violation

At present, the following pseudo-code sample illustrates the pattern detected by this rule.

```
using System;
using System.Xml;
using System.Xml.XPath;
using System.Xml.Xsl;

namespace TestForXslTransform
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create a new XslTransform object.
            XslTransform xslt = new XslTransform();

            // Load the stylesheet.
            xslt.Load("https://server/favorite.xsl");

            // Create a new XPathDocument and load the XML data to be transformed.
            XPathDocument mydata = new XPathDocument("inputdata.xml");

            // Create an XmlTextWriter which outputs to the console.
            XmlWriter writer = new XmlTextWriter(Console.Out);

            // Transform the data and send the output to the console.
            xslt.Transform(mydata, null, writer, null);
        }
    }
}
```

Solution

```
using System.Xml;
using System.Xml.Xsl;

namespace TestForXslTransform
{
    class Program
    {
        static void Main(string[] args)
        {
            // Default XsltSettings constructor disables the XSLT document() function
            // and embedded script blocks.
            XsltSettings settings = new XsltSettings();

            // Execute the transform.
            XslCompiledTransform xslt = new XslCompiledTransform();
            xslt.Load("https://server/favorite.xsl", settings, new XmlUrlResolver());
            xslt.Transform("inputdata.xml", "outputdata.html");
        }
    }
}
```

CA5375: Do not use account shared access signature

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5375
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Generating an account Shared Access Signature (SAS) with the `GetSharedAccessSignature` method under the `Microsoft.WindowsAzure.Storage` namespace.

Rule description

An account SAS can delegate access to read, write, and delete operations on blob containers, tables, queues, and file shares that are not permitted with a service SAS. However, it doesn't support container-level policies and has less flexibility and control over the permissions that are granted. If possible, use a service SAS for fine grained access control. For more information, see Delegate access with a shared access signature.

How to fix violations

Use a service SAS instead of an account SAS for fine grained access control and container-level access policy.

When to suppress warnings

It is safe to suppress this rule if you're sure that the permissions of all resources are as restricted as possible.

Pseudo-code examples

Violation

At present, the following pseudo-code sample illustrates the pattern detected by this rule.

```
using System;
using Microsoft.WindowsAzure.Storage;

class ExampleClass
{
    public void ExampleMethod(SharedAccessAccountPolicy policy)
    {
        CloudStorageAccount cloudStorageAccount = new CloudStorageAccount();
        cloudStorageAccount.GetSharedAccessSignature(policy);
    }
}
```

Solution

Instead of account SAS, use service SAS.

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.File;

class ExampleClass
{
    public void ExampleMethod(StorageCredentials storageCredentials, SharedAccessFilePolicy policy,
SharedAccessFileHeaders headers, string groupPolicyIdentifier, IPAddressOrRange ipAddressOrRange)
    {
        CloudFile cloudFile = new CloudFile(storageCredentials);
        SharedAccessProtocol protocols = SharedAccessProtocol.HttpsOnly;
        cloudFile.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

Related rules

CA5376: Use SharedAccessProtocol HttpsOnly

CA5377: Use container level access policy

CA5376: Use SharedAccessProtocol HttpsOnly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5376
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using the `GetSharedAccessSignature` method under the `Microsoft.WindowsAzure.Storage` namespace to generate a Shared Access Signature (SAS) with specifying `protocols` as `HttpsOrHttp`.

Rule description

SAS is a sensitive data which can't be transported in plain text on HTTP.

How to fix violations

Using `HttpsOnly` when generating SAS.

When to suppress warnings

Do not suppress this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5376.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5376.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5376.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5376.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5376.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

At present, the following pseudo-code sample illustrates the pattern detected by this rule.

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.File;

class ExampleClass
{
    public void ExampleMethod(SharedAccessFilePolicy policy, SharedAccessFileHeaders headers, string
groupPolicyIdentifier, IPAddressOrRange ipAddressOrRange)
    {
        CloudFile cloudFile = new CloudFile(null);
        SharedAccessProtocol protocols = SharedAccessProtocol.HttpsOrHttp;
        cloudFile.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

Solution

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.File;

class ExampleClass
{
    public void ExampleMethod(SharedAccessFilePolicy policy, SharedAccessFileHeaders headers, string
groupPolicyIdentifier, IPAddressOrRange ipAddressOrRange)
    {
        CloudFile cloudFile = new CloudFile(null);
        SharedAccessProtocol protocols = SharedAccessProtocol.HttpsOnly;
        cloudFile.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

Related rules

CA5375: Do not use account shared access signature

CA5377: Use container level access policy

CA5377: Use container level access policy

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5377
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Container level policy is not set when generating a service Shared Access Signature (SAS).

Rule description

A container-level access policy can be modified or revoked at any time. It provides greater flexibility and control over the permissions that are granted. For more information, see Define a stored access policy.

How to fix violations

Specify a valid group policy identifier when generating the service SAS.

When to suppress warnings

It is safe to suppress this rule if you're sure that the permissions of all resources are as restricted as possible.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5377.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5377.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5377.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5377.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5377.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

At present, the following pseudo-code sample illustrates the pattern detected by this rule.

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Blob;

class ExampleClass
{
    public void ExampleMethod(SharedAccessBlobPolicy policy, SharedAccessBlobHeaders headers,
Nullable<SharedAccessProtocol> protocols, IPAddressOrRange ipAddressOrRange)
    {
        var cloudAppendBlob = new CloudAppendBlob(null);
        string groupPolicyIdentifier = null;
        cloudAppendBlob.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

Solution

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Blob;

class ExampleClass
{
    public void ExampleMethod(SharedAccessBlobPolicy policy, SharedAccessBlobHeaders headers,
Nullable<SharedAccessProtocol> protocols, IPAddressOrRange ipAddressOrRange)
    {
        CloudAppendBlob cloudAppendBlob = new CloudAppendBlob(null);
        string groupPolicyIdentifier = "123";
        cloudAppendBlob.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

Related rules

CA5375: Do not use account shared access signature

CA5376: Use SharedAccessProtocol HttpsOnly

CA5378: Do not disable ServicePointManagerSecurityProtocols

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5378
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

A AppContext.SetSwitch method call sets `Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols` to `true`.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

Setting `Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols` to `true` limits Windows Communication Framework's (WCF) Transport Layer Security (TLS) connections to using TLS 1.0. That version of TLS will be deprecated. For more information, see Transport Layer Security (TLS) best practices with the .NET Framework.

How to fix violations

- If your application targets .NET Framework v4.7 or later, you can either remove the AppContext.SetSwitch method call, or set the switch's value to `false`.
- If your application targets .NET Framework v4.6.2 or earlier and runs on .NET Framework v4.7 or later, set the switch's value to `false`.
- Otherwise, refer to Transport Layer Security (TLS) best practices with the .NET Framework for mitigations.

When to suppress warnings

You can suppress this warning if you need to connect to a legacy service that can't be upgraded to use secure TLS configurations.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5378.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5378.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5378.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5378.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5378.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5378 violation
        AppContext.SetSwitch("Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols",
true);
    }
}
```

```
Imports System

Public Class ExampleClass
    Public Sub ExampleMethod()
        ' CA5378 violation
        AppContext.SetSwitch("Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols",
true)
    End Sub
End Class
```

Solution

```
using System;

public class ExampleClass
{
    public void ExampleMethod()
    {
        AppContext.SetSwitch("Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols",
false);
    }
}
```

```
Imports System

Public Class ExampleClass
    Public Sub ExampleMethod()
        AppContext.SetSwitch("Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols",
false)
    End Sub
End Class
```

CA5379: Do not use weak key derivation function algorithm

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5379
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using System.Security.Cryptography.MD5 or System.Security.Cryptography.SHA1 when instantiating System.Security.Cryptography.Rfc2898DeriveBytes.

Rule description

The Rfc2898DeriveBytes class defaults to using the SHA1 algorithm. When instantiating an Rfc2898DeriveBytes object, you should specify a hash algorithm of SHA256 or higher. Note that Rfc2898DeriveBytes.HashAlgorithm property only has a `get` accessor.

How to fix violations

Because MD5 or SHA1 are vulnerable to collisions, use SHA256 or higher for the Rfc2898DeriveBytes class.

When to suppress warnings

It is not recommended to suppress this rule except for application compatibility reasons.

Pseudo-code examples

Specify hash algorithm in constructor violation

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt, iterations, HashAlgorithmName.MD5);
    }
}
```

Specify hash algorithm in derived class' constructor violation

```
using System.Security.Cryptography;

class DerivedClass : Rfc2898DeriveBytes
{
    public DerivedClass (byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm) :
base(password, salt, iterations, hashAlgorithm)
    {
    }
}

class ExampleClass
{
    public void ExampleMethod(byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm)
    {
        var derivedClass = new DerivedClass(password, salt, iterations, HashAlgorithmName.MD5);
    }
}
```

Set hash algorithm property in derived classes violation

```
using System.Security.Cryptography;

class DerivedClass : Rfc2898DeriveBytes
{
    public DerivedClass (byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm) :
base(password, salt, iterations, hashAlgorithm)
    {
    }

    public HashAlgorithmName HashAlgorithm { get; set;}
}

class ExampleClass
{
    public void ExampleMethod(byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm)
    {
        var derivedClass = new DerivedClass(password, salt, iterations, HashAlgorithmName.MD5);
        derivedClass.HashAlgorithm = HashAlgorithmName.SHA256;
    }
}
```

Solution

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt, iterations, HashAlgorithmName.SHA256);
    }
}
```

CA5380: Do not add certificates to root store

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5380
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Adding certificates to the operating system's trusted root certificates increases the risk of legitimizing untrusted certification authority.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

This rule detects code that adds a certificate into the Trusted Root Certification Authorities certificate store. By default, the Trusted Root Certification Authorities certificate store is configured with a set of public CAs that has met the requirements of the Microsoft Root Certificate Program. Since all trusted root certification authorities (CA's) can issue certificates for any domain, an attacker can pick a weak or coercible CA that you install by yourself to target for an attack – and a single vulnerable, malicious or coercible CA undermines the security of the entire system.

How to fix violations

Do not install certificates into the Trusted Root Certification Authorities certificate store.

When to suppress warnings

It is not recommended to suppress this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5380.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)

- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5380.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5380.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5380.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5380.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule.

```
using System.Security.Cryptography.X509Certificates;

class TestClass
{
    public void TestMethod()
    {
        var storeName = StoreName.Root;
        var x509Store = new X509Store(storeName);
        x509Store.Add(new X509Certificate2());
    }
}
```

Solution

```
using System.Security.Cryptography.X509Certificates;

class TestClass
{
    public void TestMethod()
    {
        var storeName = StoreName.My;
        var x509Store = new X509Store(storeName);
        x509Store.Add(new X509Certificate2());
    }
}
```

CA5381: Ensure certificates are not added to root store

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5381
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Adding certificates to the operating system's trusted root certificates increases the risk of legitimizing untrusted certification authority.

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

This rule detects code that potentially adds a certificate into the Trusted Root Certification Authorities certificate store. By default, the Trusted Root Certification Authorities certificate store is configured with a set of public certification authorities (CAs) that has met the requirements of the Microsoft Root Certificate Program. Since all trusted root CAs can issue certificates for any domain, an attacker can pick a weak or coercible CA that you install by yourself to target for an attack – and a single vulnerable, malicious or coercible CA undermines the security of the entire system.

How to fix violations

Do not install certificates into the Trusted Root Certification Authorities certificate store.

When to suppress warnings

It is not recommended to suppress this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5381.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5381.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5381.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5381.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5381.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule.

```
using System;
using System.Security.Cryptography.X509Certificates;

class TestClass
{
    public void TestMethod()
    {
        var storeName = StoreName.Root;
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            storeName = StoreName.My;
        }

        var x509Store = new X509Store(storeName);
        x509Store.Add(new X509Certificate2());
    }
}
```

Solution

```
using System.Security.Cryptography.X509Certificates;

class TestClass
{
    public void TestMethod()
    {
        var storeName = StoreName.My;
        var x509Store = new X509Store(storeName);
        x509Store.Add(new X509Certificate2());
    }
}
```

CA5382: Use secure cookies in ASP.NET Core

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5382
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The Microsoft.AspNetCore.Http.CookieOptions.Secure property is set as `false` when invoking Microsoft.AspNetCore.Http.IResponseCookies.Append. For now, this rule only looks at the Microsoft.AspNetCore.Http.Internal.ResponseCookies class, which is one of the implementations of IResponseCookies.

This rule is similar to CA5383, but analysis can determine that the Secure property is definitely `false` or not set.

Rule description

Applications available over HTTPS must use secure cookies, which indicate to the browser that the cookie should only be transmitted using Transport Layer Security (TLS).

How to fix violations

Set Secure property as `true`.

When to suppress warnings

- If cookies are configured to be secure by default, such as using Microsoft.AspNetCore.CookiePolicy.CookiePolicyMiddleware in `Startup.Configure`:

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseCookiePolicy(
            new CookiePolicyOptions
            {
                Secure = CookieSecurePolicy.Always
            });
    }
}
```

- If you're sure there's no sensitive data in the cookies.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5382.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5382.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5382.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5382.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5382.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule.

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Internal;

class ExampleClass
{
    public void ExampleMethod(string key, string value)
    {
        var cookieOptions = new CookieOptions();
        cookieOptions.Secure = false;
        var responseCookies = new ResponseCookies(null, null);
        responseCookies.Append(key, value, cookieOptions);
    }
}
```

Solution

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Internal;

class ExampleClass
{
    public void ExampleMethod(string key, string value)
    {
        var cookieOptions = new CookieOptions();
        cookieOptions.Secure = true;
        var responseCookies = new ResponseCookies(null, null);
        responseCookies.Append(key, value, cookieOptions);
    }
}
```

CA5383: Ensure use secure cookies in ASP.NET Core

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5383
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The Microsoft.AspNetCore.Http.CookieOptions.Secure property may be set as `false` when invoking Microsoft.AspNetCore.Http.IResponseCookies.Append. For now, this rule only looks at the Microsoft.AspNetCore.Http.Internal.ResponseCookies class, which is one of the implementations of IResponseCookies.

This rule is similar to CA5382, but analysis can't determine that the Secure property is definitely `false` or not set.

Rule description

Applications available over HTTPS must use secure cookies, which indicate to the browser that the cookie should only be transmitted using Transport Layer Security (TLS).

How to fix violations

Set Secure property as `true` under all circumstances.

When to suppress warnings

- If cookies are configured to be secure by default, such as using Microsoft.AspNetCore.CookiePolicy.CookiePolicyMiddleware in `Startup.Configure`:

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseCookiePolicy(
            new CookiePolicyOptions
            {
                Secure = CookieSecurePolicy.Always
            });
    }
}
```

- If you're sure there's no sensitive data in the cookies.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5383.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5383.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5383.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5383.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5383.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule.

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Internal;

class ExampleClass
{
    public void ExampleMethod(string key, string value)
    {
        var cookieOptions = new CookieOptions();
        cookieOptions.Secure = false;
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            cookieOptions.Secure = true;
        }

        var responseCookies = new ResponseCookies(null, null);
        responseCookies.Append(key, value, cookieOptions);
    }
}
```

Solution

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Internal;

class ExampleClass
{
    public void ExampleMethod(string key, string value)
    {
        var cookieOptions = new CookieOptions();
        cookieOptions.Secure = true;
        var responseCookies = new ResponseCookies(null, null);
        responseCookies.Append(key, value, cookieOptions);
    }
}
```

CA5384: Do not use digital signature algorithm (DSA)

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5384
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using DSA in one of the following ways:

- Returning or instantiating derived classes of System.Security.Cryptography.DSA
- Using System.Security.Cryptography.AsymmetricAlgorithm.Create or System.Security.Cryptography.CryptoConfig.CreateFromName to create a DSA object.

Rule description

DSA is a weak asymmetric encryption algorithm.

How to fix violations

Switch to an RSA with at least 2048 key size, ECDH or ECDsa algorithm instead.

When to suppress warnings

It is not recommended to suppress this rule unless for compatibility with legacy applications and data.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule.

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        DSACng dsaCng = new DSACng();
    }
}
```

Solution

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        AsymmetricAlgorithm asymmetricAlgorithm = AsymmetricAlgorithm.Create("ECDsa");
    }
}
```

CA5385: Use Rivest–Shamir–Adleman (RSA) algorithm with sufficient key size

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5385
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using asymmetric encryption algorithm RSA with key size less than 2048 in one of the following ways:

- Instantiating any descendant classes of System.Security.Cryptography.RSA and specifying the `KeySize` parameter as less than 2048.
- Returning any object whose type is descendant of System.Security.Cryptography.RSA.
- Using System.Security.Cryptography.AsymmetricAlgorithm.Create without parameter which would create RSA with the default key size 1024.
- Using System.Security.Cryptography.AsymmetricAlgorithm.Create and specifying the `algName` parameter as `RSA` with the default key size 1024.
- Using System.Security.Cryptography.CryptoConfig.CreateFromName and specifying the `name` parameter as `RSA` with the default key size 1024.
- Using System.Security.Cryptography.CryptoConfig.CreateFromName and specifying the `name` parameter as `RSA` and specifying the key size as smaller than 2048 explicitly by `args`.

Rule description

An RSA key smaller than 2048 bits is more vulnerable to brute force attacks.

How to fix violations

Switch to an RSA with at least 2048 key size, ECDH or ECDsa algorithm instead.

When to suppress warnings

It is not recommended to suppress this rule unless for compatibility with legacy applications and data.

Pseudo-code examples

Violation

The following pseudo-code sample illustrates the pattern detected by this rule.

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        RSACng rsaCng = new RSACng(1024);
    }
}
```

Solution

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        RSACng rsaCng = new RSACng(2048);
    }
}
```

CA5386: Avoid hardcoding SecurityProtocolType value

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5386
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

This rule fires when either of the following conditions are met:

- A safe but hardcoded System.Net.SecurityProtocolType value was referenced.
- An integer value representing a safe protocol version was assigned to a SecurityProtocolType variable.

Safe values are:

- Tls12
- Tls13

Rule description

Transport Layer Security (TLS) secures communication between computers, most commonly with Hypertext Transfer Protocol Secure (HTTPS). Protocol versions TLS 1.0 and TLS 1.1 are deprecated, while TLS 1.2 and TLS 1.3 are current. In the future, TLS 1.2 and TLS 1.3 may be deprecated. To ensure that your application remains secure, avoid hardcoding a protocol version and target at least .NET Framework v4.7.1. For more information, see Transport Layer Security (TLS) best practices with the .NET Framework.

How to fix violations

Don't hardcode TLS protocol versions.

When to suppress warnings

You can suppress this warning if your application targets .NET Framework v4.6.2 or earlier and may run on a computer that has insecure defaults.

Pseudo-code examples

Enumeration name violation

```
using System;
using System.Net;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5386 violation
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5386 violation
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12
    End Sub
End Class
```

Integer value violation

```
using System;
using System.Net;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5386 violation
        ServicePointManager.SecurityProtocol = (SecurityProtocolType) 3072;    // TLS 1.2
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5386 violation
        ServicePointManager.SecurityProtocol = CType(3072, SecurityProtocolType)   ' TLS 1.2
    End Sub
End Class
```

Solution

```
using System;
using System.Net;

public class TestClass
{
    public void TestMethod()
    {
        // Let the operating system decide what TLS protocol version to use.
        // See https://docs.microsoft.com/dotnet/framework/network-programming/tls
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' Let the operating system decide what TLS protocol version to use.
        ' See https://docs.microsoft.com/dotnet/framework/network-programming/tls
    End Sub
End Class
```

Related rules

CA5364: Do not use deprecated security protocols

CA5397: Do not use deprecated SslProtocols values

CA5398: Avoid hardcoded SslProtocols values

CA5387: Do not use weak key derivation function with insufficient iteration count

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5387
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using System.Security.Cryptography.Rfc2898DeriveBytes.GetBytes with the default iteration count or specifying an iteration count of less than 100,000.

Rule description

This rule checks if a cryptographic key was generated by Rfc2898DeriveBytes with an iteration count of less than 100,000. A higher iteration count can help mitigate against dictionary attacks that try to guess the generated cryptographic key.

This rule is similar to CA5388, but analysis determines that the iteration count is less than 100,000.

How to fix violations

Set the iteration count greater than or equal with 100,000 before calling GetBytes.

When to suppress warnings

If you need to use a smaller iteration count for compatibility with existing data.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5387.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.

- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5387.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5387.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5387.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5387.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Default iteration count violation

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

Specify iteration count in constructor violation

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt, 100);
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

Specify iteration count by property assignment violation

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);
        rfc2898DeriveBytes.IterationCount = 100;
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

Solution

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);
        rfc2898DeriveBytes.IterationCount = 100000;
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

CA5388: Ensure sufficient iteration count when using weak key derivation function

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5388
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Iteration count may be smaller than 100,000 when deriving cryptographic key by System.Security.Cryptography.Rfc2898DeriveBytes.GetBytes.

Rule description

This rule checks if a cryptographic key was generated by Rfc2898DeriveBytes with an iteration count that may be less than 100,000. A higher iteration count can help mitigate against dictionary attacks that try to guess the generated cryptographic key.

This rule is similar to CA5387, but analysis can't determine if the iteration count is less than 100,000.

How to fix violations

Set the iteration count greater than or equal with 100k before calling GetBytes explicitly.

When to suppress warnings

It's safe to suppress warnings from this rule if:

- You need to use a smaller iteration count for compatibility with existing data.
- You're sure that the iteration count is set above 100,000.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5388.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5388.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5388.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5388.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5388.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var iterations = 100;
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            iterations = 100000;
        }

        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt, iterations);
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

Solution

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);
        rfc2898DeriveBytes.IterationCount = 100000;
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

CA5389: Do not add archive item's path to the target file system path

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5389
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Non-sanitized source file path is used as the target file path in one of these parameters:

- parameter `destinationFileName` of method ZipFileExtensions.ExtractToFile
- parameter path of method File.Open
- parameter path of method File.OpenWrite
- parameter path of method File.Create
- parameter path of constructor System.IO.FileStream
- parameter path of `fileName` System.IO.FileInfo

By default, this rule analyzes the entire codebase, but this is configurable.

Rule description

File path can be relative and can lead to file system access outside of the expected file system target path, leading to malicious config changes and remote code execution via lay-and-wait technique.

How to fix violations

Do not use the source file path to construct the target file path, or make sure that the last character on the extraction path is the directory separator character.

When to suppress warnings

You can suppress this warning if the source path always comes from a trusted source.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5389.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5389.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5389.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5389.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5389.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Violation

As of the time of this writing, the following pseudo-code sample illustrates the pattern detected by this rule.

```
using System.IO.Compression;

class TestClass
{
    public void TestMethod(ZipArchiveEntry zipArchiveEntry)
    {
        zipArchiveEntry.ExtractToFile(zipArchiveEntry.FullName);
    }
}
```

Solution

```
using System;
using System.IO;
using System.IO.Compression;

class Program
{
    static void Main(string[] args)
    {
        string zipPath = @".\result.zip";

        Console.WriteLine("Provide path where to extract the zip file:");
        string extractPath = Console.ReadLine();

        // Normalizes the path.
        extractPath = Path.GetFullPath(extractPath);

        // Ensures that the last character on the extraction path
        // is the directory separator char.
        // Without this, a malicious zip file could try to traverse outside of the expected
        // extraction path.
        if (!extractPath.EndsWith(Path.DirectorySeparatorChar.ToString(), StringComparison.Ordinal))
            extractPath += Path.DirectorySeparatorChar;

        using (ZipArchive archive = ZipFile.OpenRead(zipPath))
        {
            foreach (ZipArchiveEntry entry in archive.Entries)
            {
                if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
                {
                    // Gets the full path to ensure that relative segments are removed.
                    string destinationPath = Path.GetFullPath(Path.Combine(extractPath, entry.FullName));

                    // Ordinal match is safest, case-sensitive volumes can be mounted within volumes that
                    // are case-insensitive.
                    if (destinationPath.StartsWith(extractPath, StringComparison.Ordinal))
                        entry.ExtractToFile(destinationPath);
                }
            }
        }
    }
}
```

CA5390: Do not hard-code encryption key

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5390
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The `key` parameter of the System.Security.Cryptography.AesCcm or System.Security.Cryptography.AesGcm constructor, System.Security.Cryptography.SymmetricAlgorithm.Key property, or the `rgbKey` parameter of the System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor or System.Security.Cryptography.SymmetricAlgorithm.CreateDecryptor method is hard-coded by one of the following:

- Byte array.
- System.Convert.FromBase64String.
- All the overloads of System.Text.Encoding.GetBytes.

Rule description

For a symmetric algorithm to be successful, the secret key must be known only to the sender and the receiver. When a key is hard-coded, it is easily discovered. Even with compiled binaries, it is easy for malicious users to extract it. Once the private key is compromised, the cipher text can be decrypted directly and is not protected anymore.

How to fix violations

- Consider redesigning your application to use a secure key management system, such as Azure Key Vault.
- Keep credentials and keys in a secure location separate from your source code.

When to suppress warnings

Do not suppress a warning from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5390.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5390.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5390.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5390.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5390.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

Hard-coded byte array violation

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] someOtherBytesForIV)
    {
        byte[] rgbKey = new byte[] {1, 2, 3};
        SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
        rijn.CreateEncryptor(rgbKey, someOtherBytesForIV);
    }
}
```

Hard-coded Convert.FromBase64String violation

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] someOtherBytesForIV)
    {
        byte[] key = Convert.FromBase64String("AAAAAaazaoensuth");
        SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
        rijn.CreateEncryptor(key, someOtherBytesForIV);
    }
}
```

Hard-coded Encoding.GetBytes violation

```
using System.Text;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] someOtherBytesForIV)
    {
        byte[] key = Encoding.ASCII.GetBytes("AAAAAaazaoensuth");
        SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
        rijn.CreateEncryptor(key, someOtherBytesForIV);
    }
}
```

Solution

```
using System.Text;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(char[] chars, byte[] someOtherBytesForIV)
    {
        byte[] key = Encoding.ASCII.GetBytes(chars);
        SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
        rijn.CreateEncryptor(key, someOtherBytesForIV);
    }
}
```

CA5391: Use antiforgery tokens in ASP.NET Core MVC controllers

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5391
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Actions that result in modifying operations don't have an antiforgery token attribute. Or, using a global antiforgery token filter without calling expected anti forgery token functions.

Rule description

Handling a `POST`, `PUT`, `PATCH`, or `DELETE` request without validating an antiforgery token may be vulnerable to cross-site request forgery attacks. A cross-site request forgery attack can send malicious requests from an authenticated user to your ASP.NET Core MVC controller.

How to fix violations

- Mark the modifying action with a valid antiforgery token attribute:
  - Microsoft.AspNetCore.Mvc.ValidateAntiForgeryTokenAttribute.
  - Attribute whose name is like `%Validate%Anti_orgery%Attribute`.
- Add the valid forgery token attribute into the global filter with Microsoft.AspNetCore.Mvc.Filters.FilterCollection.Add.
- Add any custom or Mvc-provided antiforgery filter class that calls `Validate` on any class that implements the Microsoft.AspNetCore.Antiforgery.IAntiforgery interface.

When to suppress warnings

It's safe to suppress this rule if solutions other than using antiforgery token attributes are adopted to mitigate CSRF vulnerabilities. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure whether the rule applies only to derived classes of Microsoft.AspNetCore.Mvc.Controller in your codebase. For example, to specify that the rule should not run on any code within derived types of ControllerBase, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5391.exclude_aspnet_core_mvc_controllerbase = true
```

Pseudo-code examples

Without anti forgery token attribute violation

```
using Microsoft.AspNetCore.Mvc;

class ExampleController : Controller
{
    [HttpDelete]
    public IActionResult ExampleAction (string actionName)
    {
        return null;
    }

    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult AnotherAction (string actionName)
    {
        return null;
    }
}
```

Without valid global anti forgery filter

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

class ExampleController : Controller
{
    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult AnotherAction (string actionName)
    {
        return null;
    }

    [HttpDelete]
    public IActionResult ExampleAction (string actionName)
    {
        return null;
    }
}

class FilterClass : IAsyncAuthorizationFilter
{
    public Task OnAuthorizationAsync (AuthorizationFilterContext context)
    {
        return null;
    }
}

class BlahClass
{
    public static void BlahMethod ()
    {
        FilterCollection filterCollection = new FilterCollection ();
        filterCollection.Add(typeof(FilterClass));
    }
}
```

Marked with an anti forgery token attribute solution

```
using Microsoft.AspNetCore.Mvc;

class ExampleController : Controller
{
    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult ExampleAction ()
    {
        return null;
    }

    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult AnotherAction ()
    {
        return null;
    }
}
```

Using an valid global anti forgery filter

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

class ExampleController : Controller
{
    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult AnotherAction()
    {
        return null;
    }

    [HttpDelete]
    public IActionResult ExampleAction()
    {
        return null;
    }
}

class FilterClass : IAsyncAuthorizationFilter
{
    private readonly IAntiforgery antiforgery;

    public FilterClass(IAntiforgery antiforgery)
    {
        this.antiforgery = antiforgery;
    }

    public Task OnAuthorizationAsync(AuthorizationFilterContext context)
    {
        return antiforgery.ValidateRequestAsync(context.HttpContext);
    }
}

class BlahClass
{
    public static void BlahMethod()
    {
        FilterCollection filterCollection = new FilterCollection();
        filterCollection.Add(typeof(FilterClass));
    }
}
```

CA5392: Use DefaultDllImportSearchPaths attribute for P/Invokes

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5392
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The DefaultDllImportSearchPathsAttribute is not specified for a Platform Invoke (P/Invoke) function.

Rule description

By default, P/Invoke functions using DllImportAttribute probe a number of directories, including the current working directory for the library to load. This can be a security issue for certain applications, leading to DLL hijacking.

For example, if a malicious DLL with the same name as the imported one is placed under the current working directory, which will be searched firstly by default, then the malicious DLL could be loaded.

For more information, see Load Library Safely.

How to fix violations

Use DefaultDllImportSearchPathsAttribute to specify the DLL search path explicitly for the assembly or the method.

When to suppress warnings

It's safe to suppress this rule if:

- You're sure the loaded assembly is what you want. For example, your application runs on a trusted server and you completely trust the files.
- The imported assembly is a commonly used system assembly, like user32.dll, and the search path strategy follows the Known DLLs mechanism.

Pseudo-code examples

```
using System;
using System.Runtime.InteropServices;

class ExampleClass
{
    [DllImport("The3rdAssembly.dll")]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);

    public void ExampleMethod()
    {
        MessageBox(new IntPtr(0), "Hello World!", "Hello Dialog", 0);
    }
}
```

Solution

```
using System;
using System.Runtime.InteropServices;

class ExampleClass
{
    [DllImport("The3rdAssembly.dll")]
    [DefaultDllImportSearchPaths(DllImportSearchPath.UserDirectories)]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);

    public void ExampleMethod()
    {
        MessageBox(new IntPtr(0), "Hello World!", "Hello Dialog", 0);
    }
}
```

Related rules

CA5393: Do not use unsafe DllImportSearchPath value

CA5393: Do not use unsafe DllImportSearchPath value

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5393
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using one of the unsafe values of System.Runtime.InteropServices.DllImportSearchPath:

- `AssemblyDirectory`
- `UseDllDirectoryForDependencies`
- `ApplicationDirectory`

Rule description

There could be a malicious DLL in the default DLL search directories and assembly directories. Or, depending on where your application is run from, there could be a malicious DLL in the application's directory.

For more information, see Load Library Safely.

How to fix violations

Use safe values of DllImportSearchPath to specify an explicit search path instead:

- `LegacyBehavior`
- `SafeDirectories`
- `System32`
- `UserDirectories`

When to suppress warnings

It's safe to suppress this rule if:

- You're sure the loaded assembly is what you want.
- The imported assembly is a commonly used system assembly, like user32.dll, and the search path strategy follows the Known DLLs mechanism.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Unsafe search path

You can configure which value of DllImportSearchPath is unsafe for the analysis. For example, to specify that the code should not use `AssemblyDirectory`, `UseDllDirectoryForDependencies` or `ApplicationDirectory`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5393.unsafe_DllImportSearchPath_bits = 770
```

You should specify the integer value of a bitwise combination of the enumeration's values.

Pseudo-code examples

```
using System;
using System.Runtime.InteropServices;

class ExampleClass
{
    [DllImport("The3rdAssembly.dll")]
    [DefaultDllImportSearchPaths(DllImportSearchPath.AssemblyDirectory)]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);

    public void ExampleMethod()
    {
        MessageBox(new IntPtr(0), "Hello World!", "Hello Dialog", 0);
    }
}
```

Solution

```
using System;
using System.Runtime.InteropServices;

class ExampleClass
{
    [DllImport("The3rdAssembly.dll")]
    [DefaultDllImportSearchPaths(DllImportSearchPath.UserDirectories)]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);

    public void ExampleMethod()
    {
        MessageBox(new IntPtr(0), "Hello World!", "Hello Dialog", 0);
    }
}
```

Related rules

CA5392: Use DefaultDllImportSearchPaths attribute for P/Invokes

CA5394: Do not use insecure randomness

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5394
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

One of the methods of System.Random is invoked.

Rule description

Using a cryptographically weak pseudo-random number generator may allow an attacker to predict what security-sensitive value will be generated.

How to fix violations

If you need an unpredictable value for security, use a cryptographically strong random number generator like System.Security.Cryptography.RandomNumberGenerator or System.Security.Cryptography.RNGCryptoServiceProvider.

When to suppress warnings

It's safe to suppress warnings from this rule if you're sure that the weak pseudo-random numbers aren't used in a security-sensitive manner.

Pseudo-code examples

Violation

```
using System;

class ExampleClass
{
    public void ExampleMethod(Random random)
    {
        var sensitiveVariable = random.Next();
    }
}
```

Solution

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(RandomNumberGenerator randomNumberGenerator, int toExclusive)
    {
        var sensitiveVariable = randomNumberGenerator.GetInt32(toExclusive);
    }
}
```

CA5395: Miss HttpVerb attribute for action methods

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5395
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Not specifying the kind of HTTP request explicitly for action methods.

Rule description

All the action methods that create, edit, delete, or otherwise modify data needs to be protected with the antiforgery attribute from cross-site request forgery attacks. Performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

How to fix violations

Mark the action methods with `HttpVerb` attribute.

When to suppress warnings

It's safe to suppress warnings from this rule if:

- You're sure that no modifying operation is taking place in the action method. Or, it's not an action method at all.
- Solutions other than using antiforgery token attributes are adopted to mitigate CSRF vulnerabilities. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Pseudo-code examples

Violation

```
using Microsoft.AspNetCore.Mvc;

[ValidateAntiForgeryToken]
class BlahController : Controller
{
}

class ExampleController : Controller
{
    public IActionResult ExampleAction()
    {
        return null;
    }
}
```

Solution

```
using Microsoft.AspNetCore.Mvc;

[ValidateAntiForgeryToken]
class BlahController : Controller
{
}

class ExampleController : Controller
{
    [HttpGet]
    public IActionResult ExampleAction()
    {
        return null;
    }
}
```

CA5396: Set HttpOnly to true for HttpCookie

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5396
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

System.Web.HttpCookie.HttpOnly is set to `false`. Note, the default value of this property is `false`.

Rule description

As a defense in depth measure, ensure security sensitive HTTP cookies are marked as HttpOnly. This indicates web browsers should disallow scripts from accessing the cookies. Injected malicious scripts are a common way of stealing cookies.

How to fix violations

Set System.Web.HttpCookie.HttpOnly to `true`.

When to suppress warnings

- If the global value of HttpOnly is set, such as in the following example:

```
<system.web>
    ...
    <httpCookies httpOnlyCookies="true" requireSSL="true" />
</system.web>
```

- If you're sure there's no sensitive data in the cookies.

Pseudo-code examples

Violation

```
using System.Web;

class ExampleClass
{
    public void ExampleMethod()
    {
        HttpCookie httpCookie = new HttpCookie("cookieName");
        httpCookie.HttpOnly = false;
    }
}
```

Solution

```
using System.Web;

class ExampleClass
{
    public void ExampleMethod()
    {
        HttpCookie httpCookie = new HttpCookie("cookieName");
        httpCookie.HttpOnly = true;
    }
}
```

CA5397: Do not use deprecated SslProtocols values

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5397
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

This rule fires when either of the following conditions are met:

- A deprecated System.Security.Authentication.SslProtocols value was referenced.
- An integer value representing a deprecated value was either assigned to a SslProtocols variable, used as a SslProtocols return value, or used as a SslProtocols argument.

Deprecated values are:

- Ssl2
- Ssl3
- Tls
- Tls10
- Tls11

Rule description

Transport Layer Security (TLS) secures communication between computers, most commonly with Hypertext Transfer Protocol Secure (HTTPS). Older protocol versions of TLS are less secure than TLS 1.2 and TLS 1.3 and are more likely to have new vulnerabilities. Avoid older protocol versions to minimize risk. For guidance on identifying and removing deprecated protocol versions, see Solving the TLS 1.0 Problem, 2nd Edition.

How to fix violations

Don't use deprecated TLS protocol versions.

When to suppress warnings

You can suppress this warning if:

- The reference to the deprecated protocol version isn't being used to enable a deprecated version.
- You need to connect to a legacy service that can't be upgraded to use secure TLS configurations.

Pseudo-code examples

Enumeration name violation

```
using System;
using System.Security.Authentication;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5397 violation for using Tls11
        SslProtocols protocols = SslProtocols.Tls11 | SslProtocols.Tls12;
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5397 violation for using Tls11
        Dim sslProtocols As SslProtocols = SslProtocols.Tls11 Or SslProtocols.Tls12
    End Sub
End Class
```

Integer value violation

```
using System;
using System.Security.Authentication;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5397 violation
        SslProtocols sslProtocols = (SslProtocols) 768;    // TLS 1.1
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5397 violation
        Dim sslProtocols As SslProtocols = CType(768, SslProtocols)   ' TLS 1.1
    End Sub
End Class
```

Solution

```
using System;
using System.Security.Authentication;

public class TestClass
{
    public void Method()
    {
        // Let the operating system decide what TLS protocol version to use.
        // See https://docs.microsoft.com/dotnet/framework/network-programming/tls
        SslProtocols sslProtocols = SslProtocols.None;
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Sub ExampleMethod()
        ' Let the operating system decide what TLS protocol version to use.
        ' See https://docs.microsoft.com/dotnet/framework/network-programming/tls
        Dim sslProtocols As SslProtocols = SslProtocols.None
    End Sub
End Class
```

Related rules

CA5364: Do not use deprecated security protocols

CA5386: Avoid hardcoding SecurityProtocolType value

CA5398: Avoid hardcoded SslProtocols values

CA5398: Avoid hardcoded SslProtocols values

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5398
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

This rule fires when either of the following conditions are met:

- A safe but hardcoded System.Security.Authentication.SslProtocols value was referenced.
- An integer value representing a safe protocol version was either assigned to a SslProtocols variable, used as a SslProtocols return value, or used as a SslProtocols argument.

Safe values are:

- Tls12
- Tls13

Rule description

Transport Layer Security (TLS) secures communication between computers, most commonly with Hypertext Transfer Protocol Secure (HTTPS). Protocol versions TLS 1.0 and TLS 1.1 are deprecated, while TLS 1.2 and TLS 1.3 are current. In the future, TLS 1.2 and TLS 1.3 may be deprecated. To ensure that your application remains secure, avoid hardcoding a protocol version. For more information, see Transport Layer Security (TLS) best practices with the .NET Framework.

How to fix violations

Don't hardcode TLS protocol versions.

When to suppress warnings

You need to connect to a legacy service that can't be upgraded to use future TLS protocol versions.

Pseudo-code examples

Enumeration name violation

```
using System;
using System.Security.Authentication;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5398 violation
        SslProtocols sslProtocols = SslProtocols.Tls12;
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Function ExampleMethod() As SslProtocols
        ' CA5398 violation
        Return SslProtocols.Tls12
    End Function
End Class
```

Integer value violation

```
using System;
using System.Security.Authentication;

public class ExampleClass
{
    public SslProtocols ExampleMethod()
    {
        // CA5398 violation
        return (SslProtocols) 3072;    // TLS 1.2
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Function ExampleMethod() As SslProtocols
        ' CA5398 violation
        Return CType(3072, SslProtocols)   ' TLS 1.2
    End Function
End Class
```

Solution

```
using System;
using System.Security.Authentication;

public class TestClass
{
    public void Method()
    {
        // Let the operating system decide what TLS protocol version to use.
        // See https://docs.microsoft.com/dotnet/framework/network-programming/tls
        SslProtocols sslProtocols = SslProtocols.None;
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Sub ExampleMethod()
        ' Let the operating system decide what TLS protocol version to use.
        ' See https://docs.microsoft.com/dotnet/framework/network-programming/tls
        Dim sslProtocols As SslProtocols = SslProtocols.None
    End Sub
End Class
```

Related rules

CA5364: Do not use deprecated security protocols

CA5386: Avoid hardcoding SecurityProtocolType value

CA5397: Do not use deprecated SslProtocols values

CA5399: Definitely disable HttpClient certificate revocation list check

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5399
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using System.Net.Http.HttpClient while providing a platform specific handler (System.Net.Http.WinHttpHandler or System.Net.Http.HttpClientHandler) whose `CheckCertificateRevocationList` property is not set to `true` will allow revoked certificates to be accepted by the HttpClient as valid.

This rule is similar to CA5400, but analysis can determine that the `CheckCertificateRevocationList` property is definitely `false` or not set.

Rule description

A revoked certificate isn't trusted anymore. It could be used by attackers passing some malicious data or stealing sensitive data in HTTPS communication.

How to fix violations

Set the `CheckCertificateRevocationList` property to `true`.

When to suppress warnings

Do not suppress this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5399.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)

- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5399.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5399.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5399.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5399.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

```
using System.Net.Http;

class ExampleClass
{
    void ExampleMethod()
    {
        WinHttpHandler winHttpHandler = new WinHttpHandler();
        winHttpHandler.CheckCertificateRevocationList = false;
        HttpClient httpClient = new HttpClient(winHttpHandler);
    }
}
```

Solution

```
using System.Net.Http;

class ExampleClass
{
    void ExampleMethod()
    {
        WinHttpHandler winHttpHandler = new WinHttpHandler();
        winHttpHandler.CheckCertificateRevocationList = true;
        HttpClient httpClient = new HttpClient(winHttpHandler);
    }
}
```

CA5400: Ensure HttpClient certificate revocation list check is not disabled

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5400
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using System.Net.Http.HttpClient while providing a platform specific handler (System.Net.Http.WinHttpHandler or System.Net.Http.HttpClientHandler) whose `CheckCertificateRevocationList` property is possibly set to `false` will allow revoked certificates to be accepted by the HttpClient as valid.

This rule is similar to CA5400, but analysis can't determine that the `CheckCertificateRevocationList` property is definitely `false` or not set.

Rule description

A revoked certificate isn't trusted anymore. It could be used by attackers passing some malicious data or stealing sensitive data in HTTPS communication.

How to fix violations

Set the `CheckCertificateRevocationList` property to `true` explicitly.

When to suppress warnings

It's safe to suppress this rule if you're sure that the `CheckCertificateRevocationList` property is set right.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Excluded symbol names

You can configure which parts of your codebase to exclude from analysis. For example, to specify that the rule should not run on any code within types named `MyType`, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA5400.excluded_symbol_names = MyType
```

Allowed symbol name formats in the option value (separated by '|'):

- Symbol name only (includes all symbols with the name, regardless of the containing type or namespace)

- Fully qualified names in the symbol's documentation ID format. Each symbol name requires a symbol kind prefix, such as "M:" prefix for methods, "T:" prefix for types, "N:" prefix for namespaces, etc.
- `.ctor` for constructors and `.cctor` for static constructors

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA5400.excluded_symbol_names = MyType`	Matches all symbols named 'MyType' in the compilation
`dotnet_code_quality.CA5400.excluded_symbol_names = MyType1|MyType2`	Matches all symbols named either 'MyType1' or 'MyType2' in the compilation
`dotnet_code_quality.CA5400.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	Matches specific method 'MyMethod' with given fully qualified signature
`dotnet_code_quality.CA5400.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMethod2(ParamType)`	Matches specific methods 'MyMethod1' and 'MyMethod2' with respective fully qualified signature

You can configure all of these options for just this rule, for all rules, or for all rules in this category (Security). For more information, see Configure FxCop analyzers.

Pseudo-code examples

```
using System;
using System.Net.Http;

class ExampleClass
{
    void ExampleMethod(bool checkCertificateRevocationList)
    {
        WinHttpHandler winHttpHandler = new WinHttpHandler();
        winHttpHandler.CheckCertificateRevocationList = checkCertificateRevocationList;
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            winHttpHandler.CheckCertificateRevocationList = true;
        }

        HttpClient httpClient = new HttpClient(winHttpHandler);
    }
}
```

Solution

```
using System.Net.Http;

class ExampleClass
{
    void ExampleMethod()
    {
        WinHttpHandler winHttpHandler = new WinHttpHandler();
        winHttpHandler.CheckCertificateRevocationList = true;
        HttpClient httpClient = new HttpClient(winHttpHandler);
    }
}
```

CA5401: Do not use CreateEncryptor with non-default IV

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5401
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

Using System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor with non-default `rgbIV`.

Rule description

Symmetric encryption should always use a non-repeatable initialization vector to prevent dictionary attacks.

This rule is similar to CA5402, but analysis determines that the initialization vector is definitely the default.

How to fix violations

Use the default `rgbIV` value, that is, use the overload of the System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor which doesn't have any parameter.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- The `rgbIV` parameter was generated by System.Security.Cryptography.SymmetricAlgorithm.GenerateIV.
- You're sure that the `rgbIV` is really random and non-repeatable.

Pseudo-code examples

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] rgbIV)
    {
        AesCng aesCng  = new AesCng();
        aesCng.IV = rgbIV;
        aesCng.CreateEncryptor();
    }
}
```

Solution

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        AesCng aesCng  = new AesCng();
        aesCng.CreateEncryptor();
    }
}
```

CA5402: Use CreateEncryptor with the default IV

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5402
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The `rgbIV` could be non-default when using System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor.

Rule description

Symmetric encryption should always use a non-repeatable initialization vector to prevent dictionary attacks.

This rule is similar to CA5401, but analysis can't determine that the initialization vector is definitely the default.

How to fix violations

Use the default `rgbIV` value explicitly, that is, use the overload of the System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor which doesn't have any parameter.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- The `rgbIV` parameter was generated by System.Security.Cryptography.SymmetricAlgorithm.GenerateIV.
- You're sure that the `rgbIV` parameter is really random and non-repeatable.
- You're sure that the initialization vector is used.

Pseudo-code examples

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] rgbIV)
    {
        AesCng aesCng  = new AesCng();
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            aesCng.IV = rgbIV;
        }

        aesCng.CreateEncryptor();
    }
}
```

Solution

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        AesCng aesCng  = new AesCng();
        aesCng.CreateEncryptor();
    }
}
```

CA5403: Do not hard-code certificate

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA5403
Category	Microsoft.Security
Breaking change	Non-breaking

Cause

The `data` or `rawData` parameter of a X509Certificate or X509Certificate2 constructor is hard-coded by one of the following:

- Byte array.
- Char array.
- System.Convert.FromBase64String(String).
- All the overloads of System.Text.Encoding.GetBytes.

Rule description

A hard-coded certificate's private key is easily discovered. Even with compiled binaries, it is easy for malicious users to extract a hard-coded certificate's private key. Once the private key is compromised, an attacker can impersonate that certificate, and any resources or operations protected by that certificate will be available to the attacker.

How to fix violations

- Consider redesigning your application to use a secure key management system, such as Azure Key Vault.
- Keep credentials and certificates in a secure location separate from your source code.

When to suppress warnings

It's safe to suppress a warning from this rule if the hard-coded data doesn't contain the certificate's private key. For example, the data is from a `.cer` file. Hard-coding public certificate information may still create a challenge for rotating certificates as they expire or get revoked.

Pseudo-code examples

Hard-coded by byte array

```
using System.IO;
using System.Security.Cryptography.X509Certificates;

class ExampleClass
{
    public void ExampleMethod(string path)
    {
        byte[] bytes = new byte[] {1, 2, 3};
        File.WriteAllBytes(path, bytes);
        new X509Certificate2(path);
    }
}
```

Hard-coded by char array

```
using System.IO;
using System.Security.Cryptography.X509Certificates;
using System.Text;

class ExampleClass
{
    public void ExampleMethod(byte[] bytes, string path)
    {
        char[] chars = new char[] { '1', '2', '3' };
        Encoding.ASCII.GetBytes(chars, 0, 3, bytes, 0);
        File.WriteAllBytes(path, bytes);
        new X509Certificate2(path);
    }
}
```

Hard-coded by FromBase64String

```
using System;
using System.IO;
using System.Security.Cryptography.X509Certificates;

class ExampleClass
{
    public void ExampleMethod(string path)
    {
        byte[] bytes = Convert.FromBase64String("AAAAAaazaoensuth");
        File.WriteAllBytes(path, bytes);
        new X509Certificate2(path);
    }
}
```

Hard-coded by GetBytes

```
using System;
using System.IO;
using System.Security.Cryptography.X509Certificates;
using System.Text;

class ExampleClass
{
    public void ExampleMethod(string path)
    {
        byte[] bytes = Encoding.ASCII.GetBytes("AAAAAaazaoensuth");
        File.WriteAllBytes(path, bytes);
        new X509Certificate2(path);
    }
}
```

Solution

```
using System.IO;
using System.Security.Cryptography.X509Certificates;

class ExampleClass
{
    public void ExampleMethod(string path)
    {
        new X509Certificate2("Certificate.cer");
    }
}
```

Usage Warnings

8/4/2020 • 8 minutes to read • Edit Online

Usage warnings support proper usage of .NET.

In This Section

RULE	DESCRIPTION
CA1801: Review unused parameters	A method signature includes a parameter that is not used in the method body.
CA1806: Do not ignore method results	A new object is created but never used; or a method that creates and returns a new string is called and the new string is never used; or a COM or P/Invoke method returns an HRESULT or error code that is never used.
CA1816: Call GC.SuppressFinalize correctly	A method that is an implementation of Dispose does not call GC.SuppressFinalize; or a method that is not an implementation of Dispose calls GC.SuppressFinalize; or a method calls GC.SuppressFinalize and passes something other than this (Me in Visual Basic).
CA2200: Rethrow to preserve stack details	An exception is rethrown and the exception is explicitly specified in the throw statement. If an exception is rethrown by specifying the exception in the throw statement, the list of method calls between the original method that threw the exception and the current method is lost.
CA2201: Do not raise reserved exception types	This makes the original error hard to detect and debug.
CA2202: Do not dispose objects multiple times	A method implementation contains code paths that could cause multiple calls to System.IDisposable.Dispose or a Dispose equivalent (such as a Close() method on some types) on the same object.
CA2204: Literals should be spelled correctly	A literal string in a method body contains one or more words that are not recognized by the Microsoft spelling checker library.
CA2205: Use managed equivalents of Win32 API	A platform invoke method is defined and a .NET method with the equivalent functionality is available.
CA2207: Initialize value type static fields inline	A value type declares an explicit static constructor. To fix a violation of this rule, initialize all static data when it is declared and remove the static constructor.
CA2208: Instantiate argument exceptions correctly	A call is made to the default (parameterless) constructor of an exception type that is or derives from ArgumentException, or an incorrect string argument is passed to a parameterized constructor of an exception type that is or derives from ArgumentException.

RULE	DESCRIPTION
CA2211: Non-constant fields should not be visible	Static fields that are not constants or read-only are not thread-safe. Access to such a field must be carefully controlled and requires advanced programming techniques for synchronizing access to the class object.
CA2212: Do not mark serviced components with WebMethod	A method in a type that inherits from System.EnterpriseServices.ServicedComponent is marked with System.Web.Services.WebMethodAttribute. Because WebMethodAttribute and a ServicedComponent method have conflicting behavior and requirements for context and transaction flow, the behavior of the method will be incorrect in some scenarios.
CA2213: Disposable fields should be disposed	A type that implements System.IDisposable declares fields that are of types that also implement IDisposable. The Dispose method of the field is not called by the Dispose method of the declaring type.
CA2214: Do not call overridable methods in constructors	When a constructor calls a virtual method, it is possible that the constructor for the instance that invokes the method has not executed.
CA2215: Dispose methods should call base class dispose	If a type inherits from a disposable type, it must call the Dispose method of the base type from its own Dispose method.
CA2216: Disposable types should declare finalizer	A type that implements System.IDisposable, and has fields that suggest the use of unmanaged resources, does not implement a finalizer as described by Object.Finalize.
CA2217: Do not mark enums with FlagsAttribute	An externally visible enumeration is marked with FlagsAttribute, and it has one or more values that are not powers of two or a combination of the other defined values on the enumeration.
CA2218: Override GetHashCode on overriding Equals	GetHashCode returns a value, based on the current instance, that is suited for hashing algorithms and data structures such as a hash table. Two objects that are the same type and are equal must return the same hash code.
CA2219: Do not raise exceptions in exception clauses	When an exception is raised in a finally or fault clause, the new exception hides the active exception. When an exception is raised in a filter clause, the run time silently catches the exception. This makes the original error hard to detect and debug.
CA2220: Finalizers should call base class finalizer	Finalization must be propagated through the inheritance hierarchy. To guarantee this, types must call their base class Finalize method in their own Finalize method.
CA2221: Finalizers should be protected	Finalizers must use the family access modifier.
CA2222: Do not decrease inherited member visibility	Don't change the access modifier for inherited members. Changing an inherited member to private does not prevent callers from accessing the base class implementation of the method.

RULE	DESCRIPTION
CA2223: Members should differ by more than return type	Although the common language runtime allows the use of return types to differentiate between otherwise identical members, this feature is not in the Common Language Specification, nor is it a common feature of .NET programming languages.
CA2224: Override equals on overloading operator equals	A public type implements the equality operator, but does not override Object.Equals.
CA2225: Operator overloads have named alternates	An operator overload was detected, and the expected named alternative method was not found. The named alternative member provides access to the same functionality as the operator, and is provided for developers who program in languages that do not support overloaded operators.
CA2226: Operators should have symmetrical overloads	A type implements the equality or inequality operator, and does not implement the opposite operator.
CA2227: Collection properties should be read only	A writable collection property allows a user to replace the collection with a different collection. A read-only property stops the collection from being replaced but still allows the individual members to be set.
CA2228: Do not ship unreleased resource formats	Resource files that were built by using pre-release versions of the .NET might not be usable by supported versions of .NET.
CA2229: Implement serialization constructors	To fix a violation of this rule, implement the serialization constructor. For a sealed class, make the constructor private; otherwise, make it protected.
CA2230: Use params for variable arguments	A public or protected type contains a public or protected method that uses the VarArgs calling convention instead of the params keyword.
CA2231: Overload operator equals on overriding ValueType.Equals	A value type overrides `Object.Equals` but does not implement the equality operator.
CA2232: Mark Windows Forms entry points with STAThread	STAThreadAttribute indicates that the COM threading model for the application is a single-threaded apartment. This attribute must be present on the entry point of any application that uses Windows Forms; if it is omitted, the Windows components might not work correctly.
CA2233: Operations should not overflow	Arithmetic operations should not be performed without first validating the operands, to make sure that the result of the operation is not outside the range of possible values for the data types involved.
CA2234: Pass System.Uri objects instead of strings	A call is made to a method that has a string parameter whose name contains "uri", "URI", "urn", "URN", "url", or "URL". The declaring type of the method contains a corresponding method overload that has a System.Uri parameter.
CA2235: Mark all non-serializable fields	An instance field of a type that is not serializable is declared in a type that is serializable.

RULE	DESCRIPTION
CA2236: Call base class methods on ISerializable types	To fix a violation of this rule, call the base type GetObjectData method or serialization constructor from the corresponding derived type method or constructor.
CA2237: Mark ISerializable types with SerializableAttribute	To be recognized by the common language runtime as serializable, types must be marked with the SerializableAttribute attribute even if the type uses a custom serialization routine through implementation of the ISerializable interface.
CA2238: Implement serialization methods correctly	A method that handles a serialization event does not have the correct signature, return type, or visibility.
CA2239: Provide deserialization methods for optional fields	A type has a field that is marked with the System.Runtime.Serialization.OptionalFieldAttribute attribute, and the type does not provide de-serialization event handling methods.
CA2240: Implement ISerializable correctly	To fix a violation of this rule, make the GetObjectData method visible and overridable, and make sure that all instance fields are included in the serialization process or explicitly marked with the NonSerializedAttribute attribute.
CA2241: Provide correct arguments to formatting methods	The format argument passed to System.String.Format does not contain a format item that corresponds to each object argument, or vice versa.
CA2242: Test for NaN correctly	This expression tests a value against Single.Nan or Double.Nan. Use Single.IsNan(Single) or Double.IsNan(Double) to test the value.
CA2243: Attribute string literals should parse correctly	An attribute's string literal parameter does not parse correctly for a URL, a GUID, or a version.
CA2244: Do not duplicate indexed element initializations	An object initializer has more than one indexed element initializer with the same constant index. All but the last initializer are redundant.
CA2245: Do not assign a property to itself	A property was accidentally assigned to itself.
CA2246: Do not assign a symbol and its member in the same statement	Assigning a symbol and its member, that is, a field or a property, in the same statement is not recommended. It is not clear if the member access was intended to use the symbol's old value prior to the assignment or the new value from the assignment in this statement.
CA2247: Argument passed to TaskCompletionSource constructor should be TaskCreationOptions enum instead of TaskContinuationOptions enum	TaskCompletionSource has constructors that take TaskCreationOptions that control the underlying Task, and constructors that take object state that's stored in the task. Accidentally passing a TaskContinuationOptions instead of a TaskCreationOptions will result in the call treating the options as state.
CA2248: Provide correct 'enum' argument to 'Enum.HasFlag'	The enum type passed as an argument to the `HasFlag` method call is different from the calling enum type.

CA1801: Review unused parameters

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1801
Category	Microsoft.Usage
Breaking change	Non-breaking - If the member is not visible outside the assembly, regardless of the change you make. Non-breaking - If you change the member to use the parameter within its body. Breaking - If you remove the parameter and it is visible outside the assembly.

Cause

A method signature includes a parameter that's not used in the method body.

This rule does not examine the following kinds of methods:

- Methods referenced by a delegate.
- Methods used as event handlers.
- Serialization constructors (see guidelines).
- Serialization GetObjectData method.
- Methods declared with the `abstract` (`MustOverride` in Visual Basic) modifier.
- Methods declared with the `virtual` (`Overridable` in Visual Basic) modifier.
- Methods declared with the `override` (`Overrides` in Visual Basic) modifier.
- Methods declared with the `extern` (`Declare` statement in Visual Basic) modifier.

If you're using FxCop analyzers, this rule does not flag parameters that are named with the discard symbol, for example, `_`, `_1`, and `_2`. This reduces warning noise on parameters that are needed for signature requirements, for example, a method used as a delegate, a parameter with special attributes, or a parameter whose value is implicitly accessed at run time by a framework but is not referenced in code.

Rule description

Review parameters in non-virtual methods that are not used in the method body to make sure no incorrectness exists around failure to access them. Unused parameters incur maintenance and performance costs.

Sometimes, a violation of this rule can point to an implementation bug in the method. For example, the parameter should have been used in the method body. Suppress warnings of this rule if the parameter must exist because of backward compatibility.

How to fix violations

To fix a violation of this rule, remove the unused parameter (a breaking change), or use the parameter in the method body (a non-breaking change).

When to suppress warnings

It is safe to suppress a warning from this rule:

- In previously shipped code for which the fix would be a breaking change.
- For the `this` parameter in a custom extension method for Microsoft.VisualStudio.TestTools.UnitTesting.Assert. The functions in the Assert class are static, so there's no need to access the `this` parameter in the method body.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca1801.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Performance). For more information, see Configure FxCop analyzers.

Example

The following example shows two methods. One method violates the rule and the other method satisfies the rule.

```
using System;
using System.Globalization;

namespace Samples
{
    public static class TestClass
    {
        // This method violates the rule.
        public static string GetSomething(int first, int second)
        {
            return first.ToString(CultureInfo.InvariantCulture);
        }

        // This method satisfies the rule.
        public static string GetSomethingElse(int first)
        {
            return first.ToString(CultureInfo.InvariantCulture);
        }
    }
}
```

Related rules

CA1811: Avoid uncalled private code

CA1812: Avoid uninstantiated internal classes

CA1804: Remove unused locals

CA2229: Implement serialization constructors

CA1806: Do not ignore method results

8/4/2020 • 4 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1806
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

There are several possible reasons for this warning:

- A new object is created but never used.
- A method that creates and returns a new string is called and the new string is never used.
- A COM or P/Invoke method that returns a HRESULT or error code that is never used.
- A LINQ method that returns a result that is never used.

Rule description

Unnecessary object creation and the associated garbage collection of the unused object degrade performance.

Strings are immutable and methods such as String.ToUpper returns a new instance of a string instead of modifying the instance of the string in the calling method.

Ignoring HRESULT or error code can lead to unexpected behavior in error conditions or to low-resource conditions.

LINQ methods are known to not have side effects, and the result should not be ignored.

How to fix violations

If method A creates a new instance of B object that is never used, pass the instance as an argument to another method or assign the instance to a variable. If the object creation is unnecessary, remove the it.

-or-

If method A calls method B, but does not use the new string instance that the method B returns, pass the instance as an argument to another method, assign the instance to a variable. Or remove the call if it is unnecessary.

-or-

If method A calls method B, but does not use the HRESULT or error code that the method returns, use the result in a conditional statement, assign the result to a variable, or pass it as an argument to another method.

-or-

If a LINQ method A calls method B, but does not use the result, use the result in a conditional statement, assign the result to a variable, or pass it as an argument to another method.

When to suppress warnings

Do not suppress a warning from this rule unless the act of creating the object serves some purpose.

Example

The following example shows a class that ignores the result of calling String.Trim.

```
using System;

namespace Samples
{
    public class Book
    {
        private readonly string _Title;

        public Book(string title)
        {
            if (title != null)
            {
                // Violates this rule
                title.Trim();
            }

            _Title = title;
        }

        public string Title
        {
            get { return _Title; }
        }
    }
}
```

```
Imports System

Namespace Samples

    Public Class Book

        Private ReadOnly _Title As String

        Public Sub New(ByVal title As String)

            If title IsNot Nothing Then
                ' Violates this rule
                title.Trim()
            End If

            _Title = title

        End Sub

        Public ReadOnly Property Title() As String
            Get
                Return _Title
            End Get
        End Property

    End Class

End Namespace
```

```
using namespace System;

namespace Samples
{
    public ref class Book
    {
        private:
            initonly String^ _Title;

        public:
        Book(String^ title)
        {
            if (title != nullptr)
            {
                // Violates this rule
                title->Trim();
            }
                _Title = title;
            }

        property String^ Title
        {
            String^ get() { return _Title; }
        }
    };
}
```

Example

The following example fixes the previous violation by assigning the result of String.Trim back to the variable it was called on.

```
using System;

namespace Samples
{
    public class Book
    {
        private readonly string _Title;

        public Book(string title)
        {
            if (title != null)
            {
                title = title.Trim();
            }

            _Title = title;
        }

        public string Title
        {
            get { return _Title; }
        }
    }
}
```

```
Imports System

Namespace Samples

    Public Class Book

        Private ReadOnly _Title As String

        Public Sub New(ByVal title As String)

            If title IsNot Nothing Then
                title = title.Trim()
            End If

            _Title = title

        End Sub

        Public ReadOnly Property Title() As String
            Get
                Return _Title
            End Get
        End Property

    End Class

End Namespace
```

```
using namespace System;

namespace Samples
{
    public ref class Book
    {
    private:
        initonly String^ _Title;

    public:
        Book(String^ title)
        {
            if (title != nullptr)
            {
                title = title->Trim();
            }

            _Title = title;
        }

        property String^ Title
        {
            String^ get() { return _Title; }
        }
    };
}
```

Example

The following example shows a method that does not use an object that it creates.

> **NOTE**
>
> This violation cannot be reproduced in Visual Basic.

```
using namespace System;

namespace Samples
{
    public ref class Book
    {
    public:
        Book()
        {
        }

        static Book^ CreateBook()
        {
            // Violates this rule
            gcnew Book();
            return gcnew Book();
        }
    };
}
```

```
using System;

namespace Samples
{
    public class Book
    {
        public Book()
        {
        }

        public static Book CreateBook()
        {
            // Violates this rule
            new Book();
            return new Book();
        }
    }
}
```

Example

The following example fixes the previous violation by removing the unnecessary creation of an object.

```
using System;

namespace Samples
{
    public class Book
    {
        public Book()
        {
        }

        public static Book CreateBook()
        {
            return new Book();
        }
    }
}
```

```
using namespace System;

namespace Samples
{
    public ref class Book
    {
    public:
        Book()
        {
        }
        static Book^ CreateBook()
        {
            return gcnew Book();
        }
    };
}
```

CA1816: Call GC.SuppressFinalize correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA1816
Category	Microsoft. Usage
Breaking change	Non-breaking

Cause

Violations of this rule can be caused by:

- A method that is an implementation of IDisposable.Dispose and doesn't call GC.SuppressFinalize.
- A method that is not an implementation of IDisposable.Dispose and calls GC.SuppressFinalize.
- A method that calls GC.SuppressFinalize and passes something other than this (C#) or Me (Visual Basic).

Rule description

The IDisposable.Dispose method lets users release resources at any time before the object becoming available for garbage collection. If the IDisposable.Dispose method is called, it frees resources of the object. This makes finalization unnecessary. IDisposable.Dispose should call GC.SuppressFinalize so the garbage collector doesn't call the finalizer of the object.

To prevent derived types with finalizers from having to reimplement IDisposable and to call it, unsealed types without finalizers should still call GC.SuppressFinalize.

How to fix violations

To fix a violation of this rule:

- If the method is an implementation of Dispose, add a call to GC.SuppressFinalize.
- If the method is not an implementation of Dispose, either remove the call to GC.SuppressFinalize or move it to the type's Dispose implementation.
- Change all calls to GC.SuppressFinalize to pass this (C#) or Me (Visual Basic).

When to suppress warnings

Only suppress a warning from this rule if you are deliberately using GC.SuppressFinalize to control the lifetime of other objects. Don't suppress a warning from this rule if an implementation of Dispose doesn't call GC.SuppressFinalize. In this situation, failing to suppress finalization degrades performance and provides no benefits.

Example that violates CA1816

This code shows a method that calls GC.SuppressFinalize, but doesn't pass this (C#) or Me (Visual Basic). As a

result, this code violates rule CA1816.

```
Imports System
Imports System.Data.SqlClient

Namespace Samples

    Public Class DatabaseConnector
        Implements IDisposable

        Private _Connection As New SqlConnection

        Public Sub Dispose() Implements IDisposable.Dispose
            Dispose(True)
            GC.SuppressFinalize(True)   ' Violates rules
        End Sub

        Protected Overridable Sub Dispose(ByVal disposing As Boolean)
            If disposing Then
                If _Connection IsNot Nothing Then
                    _Connection.Dispose()
                    _Connection = Nothing
                End If
            End If
        End Sub

    End Class

End Namespace
```

```
using System;
using System.Data.SqlClient;
namespace Samples
{
    public class DatabaseConnector : IDisposable
    {
        private SqlConnection _Connection = new SqlConnection();

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(true);  // Violates rule
        }

        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                if (_Connection != null)
                {
                    _Connection.Dispose();
                    _Connection = null;
                }
            }
        }
    }
}
```

Example that satisfies CA1816

This example shows a method that correctly calls GC.SuppressFinalize by passing this (C#) or Me (Visual Basic).

```
Imports System
Imports System.Data.SqlClient

Namespace Samples

    Public Class DatabaseConnector
        Implements IDisposable

        Private _Connection As New SqlConnection

        Public Sub Dispose() Implements IDisposable.Dispose
            Dispose(True)
            GC.SuppressFinalize(Me)
        End Sub

        Protected Overridable Sub Dispose(ByVal disposing As Boolean)
            If disposing Then
                If _Connection IsNot Nothing Then
                    _Connection.Dispose()
                    _Connection = Nothing
                End If
            End If
        End Sub

    End Class

End Namespace
```

```
using System;
using System.Data.SqlClient;

namespace Samples
{
    public class DatabaseConnector : IDisposable
    {
        private SqlConnection _Connection = new SqlConnection();

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                if (_Connection != null)
                {
                    _Connection.Dispose();
                    _Connection = null;
                }
            }
        }
    }
}
```

Related rules

- CA2215: Dispose methods should call base class dispose
- CA2216: Disposable types should declare finalizer

See also

- Dispose pattern

CA2200: Rethrow to preserve stack details

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2200
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An exception is rethrown and the exception is explicitly specified in the `throw` statement.

Rule description

Once an exception is thrown, part of the information it carries is the stack trace. The stack trace is a list of the method call hierarchy that starts with the method that throws the exception and ends with the method that catches the exception. If an exception is re-thrown by specifying the exception in the `throw` statement, the stack trace is restarted at the current method and the list of method calls between the original method that threw the exception and the current method is lost. To keep the original stack trace information with the exception, use the `throw` statement without specifying the exception.

How to fix violations

To fix a violation of this rule, rethrow the exception without specifying the exception explicitly.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a method, `CatchAndRethrowExplicitly`, which violates the rule and a method, `CatchAndRethrowImplicitly`, which satisfies the rule.

```
using System;

namespace UsageLibrary
{
   class TestsRethrow
   {
      static void Main()
      {
         TestsRethrow testRethrow = new TestsRethrow();
         testRethrow.CatchException();
      }

      void CatchException()
      {
         try
```

```
            {
               CatchAndRethrowExplicitly();
            }
            catch(ArithmeticException e)
            {
               Console.WriteLine("Explicitly specified:{0}{1}",
                  Environment.NewLine, e.StackTrace);
            }

            try
            {
               CatchAndRethrowImplicitly();
            }
            catch(ArithmeticException e)
            {
               Console.WriteLine("{0}Implicitly specified:{0}{1}",
                  Environment.NewLine, e.StackTrace);
            }
         }

         void CatchAndRethrowExplicitly()
         {
            try
            {
               ThrowException();
            }
            catch(ArithmeticException e)
            {
               // Violates the rule.
               throw e;
            }
         }

         void CatchAndRethrowImplicitly()
         {
            try
            {
               ThrowException();
            }
            catch(ArithmeticException e)
            {
               // Satisfies the rule.
               throw;
            }
         }

         void ThrowException()
         {
            throw new ArithmeticException("illegal expression");
         }
      }
   }
```

```
Imports System

Namespace UsageLibrary

   Class TestsRethrow

      Shared Sub Main()
         Dim testRethrow As New TestsRethrow()
         testRethrow.CatchException()
      End Sub

      Sub CatchException()

         Try
            CatchAndRethrowExplicitly()
         Catch e As ArithmeticException
            Console.WriteLine("Explicitly specified:{0}{1}", _
               Environment.NewLine, e.StackTrace)
         End Try

         Try
            CatchAndRethrowImplicitly()
         Catch e As ArithmeticException
            Console.WriteLine("{0}Implicitly specified:{0}{1}", _
               Environment.NewLine, e.StackTrace)
         End Try

      End Sub

      Sub CatchAndRethrowExplicitly()

         Try
            ThrowException()
         Catch e As ArithmeticException

            ' Violates the rule.
            Throw e
         End Try

      End Sub

      Sub CatchAndRethrowImplicitly()

         Try
            ThrowException()
         Catch e As ArithmeticException

            ' Satisfies the rule.
            Throw
         End Try

      End Sub

      Sub ThrowException()
         Throw New ArithmeticException("illegal expression")
      End Sub

   End Class

End Namespace
```

CA2201: Do not raise reserved exception types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2201
Category	Microsoft.Usage
Breaking change	Breaking

Cause

A method raises an exception type that is too general or that is reserved by the runtime.

Rule description

The following exception types are too general to provide sufficient information to the user:

- System.Exception
- System.ApplicationException
- System.SystemException

The following exception types are reserved and should be thrown only by the common language runtime:

- System.AccessViolationException
- System.ExecutionEngineException
- System.IndexOutOfRangeException
- System.NullReferenceException
- System.OutOfMemoryException
- System.Runtime.InteropServices.COMException
- System.Runtime.InteropServices.ExternalException
- System.Runtime.InteropServices.SEHException
- System.StackOverflowException

Do Not Throw General Exceptions

If you throw a general exception type, such as Exception or SystemException in a library or framework, it forces consumers to catch all exceptions, including unknown exceptions that they do not know how to handle.

Instead, either throw a more derived type that already exists in the framework, or create your own type that derives from Exception.

Throw Specific Exceptions

The following table shows parameters and which exceptions to throw when you validate the parameter, including

the value parameter in the set accessor of a property:

PARAMETER DESCRIPTION	EXCEPTION
`null` reference	System.ArgumentNullException
Outside the allowed range of values (such as an index for a collection or list)	System.ArgumentOutOfRangeException
Invalid `enum` value	System.ComponentModel.InvalidEnumArgumentException
Contains a format that does not meet the parameter specifications of a method (such as the format string for `ToString(String)`)	System.FormatException
Otherwise invalid	System.ArgumentException

When an operation is invalid for the current state of an object throw System.InvalidOperationException

When an operation is performed on an object that has been disposed throw System.ObjectDisposedException

When an operation is not supported (such as in an overridden **Stream.Write** in a Stream opened for reading) throw System.NotSupportedException

When a conversion would result in an overflow (such as in a explicit cast operator overload) throw System.OverflowException

For all other situations, consider creating your own type that derives from Exception and throw that.

How to fix violations

To fix a violation of this rule, change the type of the thrown exception to a specific type that is not one of the reserved types.

When to suppress warnings

Do not suppress a warning from this rule.

Related rules

- CA1031: Do not catch general exception types

CA2202: Do not dispose objects multiple times

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2202
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A method implementation contains code paths that could cause multiple calls to System.IDisposable.Dispose or a Dispose equivalent, such as a Close() method on some types, on the same object.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

A correctly implemented Dispose method can be called multiple times without throwing an exception. However, this is not guaranteed and to avoid generating a System.ObjectDisposedException you should not call Dispose more than one time on an object.

Related rules

- CA2000: Dispose objects before losing scope

How to fix violations

To fix a violation of this rule, change the implementation so that regardless of the code path, Dispose is called only one time for the object.

When to suppress warnings

Do not suppress a warning from this rule. Even if Dispose for the object is known to be safely callable multiple times, the implementation might change in the future.

Example

Nested `using` statements (`Using` in Visual Basic) can cause violations of the CA2202 warning. If the IDisposable resource of the nested inner `using` statement contains the resource of the outer `using` statement, the `Dispose` method of the nested resource releases the contained resource. When this situation occurs, the `Dispose` method of the outer `using` statement attempts to dispose its resource for a second time.

In the following example, a Stream object that is created in an outer using statement is released at the end of the inner using statement in the Dispose method of the StreamWriter object that contains the `stream` object. At the

end of the outer `using` statement, the `stream` object is released a second time. The second release is a violation of CA2202.

```
using (Stream stream = new FileStream("file.txt", FileMode.OpenOrCreate))
{
    using (StreamWriter writer = new StreamWriter(stream))
    {
        // Use the writer object...
    }
}
```

Example

To resolve this issue, use a `try` / `finally` block instead of the outer `using` statement. In the `finally` block, make sure that the `stream` resource is not null.

```
Stream stream = null;
try
{
    stream = new FileStream("file.txt", FileMode.OpenOrCreate);
    using (StreamWriter writer = new StreamWriter(stream))
    {
        stream = null;
        // Use the writer object...
    }
}
finally
{
    stream?.Dispose();
}
```

TIP

The `?.` syntax above is the null-conditional operator.

See also

- System.IDisposable
- Dispose Pattern

CA2204: Literals should be spelled correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2204
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A literal string is passed as an argument for a localizable parameter, or to a localizable property, and the string contains one or more words that are not recognized by the Microsoft spelling checker library.

Rule description

This rule checks a literal string that is passed as a value to a parameter or property when one or more of the following cases is true:

- The LocalizableAttribute attribute of the parameter or property is set to true.
- The parameter or property name contains "Text", "Message", or "Caption".
- The name of the string variable that is passed to a Write or WriteLine() method is either "value" or "format".

This rule parses the literal string into words, tokenizing compound words, and checks the spelling of each word or token. For information about the parsing algorithm, see CA1704: Identifiers should be spelled correctly.

Language

The spell checker currently checks only against English-based culture dictionaries. You can change the culture of your project in the project file, by adding the **CodeAnalysisCulture** element.

For example:

```
<Project ...>
  <PropertyGroup>
    <CodeAnalysisCulture>en-AU</CodeAnalysisCulture>
```

> **IMPORTANT**
>
> If you set the culture to anything other than an English-based culture, this code analysis rule is silently disabled.

How to fix violations

To fix a violation of this rule, correct the spelling of the word or add the word to a custom dictionary. For information about how to use custom dictionaries, see How to: Customize the Code Analysis Dictionary.

When to suppress warnings

Do not suppress a warning from this rule. Correctly spelled words reduce the learning curve required for new software libraries.

Related rules

- CA1704: Identifiers should be spelled correctly
- CA1703: Resource strings should be spelled correctly

CA2205: Use managed equivalents of Win32 API

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2205
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A platform invoke method is defined and a method with the equivalent functionality exists in .NET.

Rule description

A platform invoke method is used to call an unmanaged DLL function and is defined using the System.Runtime.InteropServices.DllImportAttribute attribute, or the `Declare` keyword in Visual Basic. An incorrectly defined platform invoke method can lead to run-time exceptions because of issues such as a misnamed function, faulty mapping of parameter and return value data types, and incorrect field specifications, such as the calling convention and character set. If available, it is simpler and less error prone to call the equivalent managed method than to define and call the unmanaged method directly. Calling a platform invoke method can also lead to additional security issues that need to be addressed.

How to fix violations

To fix a violation of this rule, replace the call to the unmanaged function with a call to its managed equivalent.

When to suppress warnings

Suppress a warning from this rule if the suggested replacement method does not provide the needed functionality.

Example

The following example shows a platform invoke method definition that violates the rule. In addition, the calls to the platform invoke method and the equivalent managed method are shown.

```
using System;
using System.Runtime.InteropServices;
using System.Text;

namespace UsageLibrary
{
   internal class NativeMethods
   {
      private NativeMethods() {}

      // The following method definition violates the rule.
      [DllImport("kernel32.dll", CharSet = CharSet.Unicode,
          SetLastError = true)]
      internal static extern int ExpandEnvironmentStrings(
         string lpSrc, StringBuilder lpDst, int nSize);
   }

   public class UseNativeMethod
   {
      public void Test()
      {
         string environmentVariable = "%TEMP%";
         StringBuilder expandedVariable = new StringBuilder(100);

         // Call the unmanaged method.
         NativeMethods.ExpandEnvironmentStrings(
            environmentVariable,
            expandedVariable,
            expandedVariable.Capacity);

         // Call the equivalent managed method.
         Environment.ExpandEnvironmentVariables(environmentVariable);
      }
   }
}
```

```
Imports System
Imports System.Runtime.InteropServices
Imports System.Text

Namespace UsageLibrary

   Class NativeMethods

      Private Sub New()
      End Sub

      ' The following method definitions violate the rule.

      <DllImport("kernel32.dll", CharSet := CharSet.Unicode, _
         SetLastError := True)> _
      Friend Shared Function ExpandEnvironmentStrings _
         (lpSrc As String, lpDst As StringBuilder, nSize As Integer) _
         As Integer
      End Function

      Friend Declare Unicode Function ExpandEnvironmentStrings2 _
         Lib "kernel32.dll" Alias "ExpandEnvironmentStrings" _
         (lpSrc As String, lpDst As StringBuilder, nSize As Integer) _
         As Integer

   End Class

   Public Class UseNativeMethod

      Shared Sub Main()

         Dim environmentVariable As String = "%TEMP%"
         Dim expandedVariable As New StringBuilder(100)

         ' Call the unmanaged method.
         NativeMethods.ExpandEnvironmentStrings( _
            environmentVariable, _
            expandedVariable, _
            expandedVariable.Capacity)

         ' Call the unmanaged method.
         NativeMethods.ExpandEnvironmentStrings2( _
            environmentVariable, _
            expandedVariable, _
            expandedVariable.Capacity)

         ' Call the equivalent managed method.
         Environment.ExpandEnvironmentVariables(environmentVariable)

      End Sub

   End Class

End Namespace
```

Related rules

- CA1404: Call GetLastError immediately after P/Invoke
- CA1060: Move P/Invokes to NativeMethods class
- CA1400: P/Invoke entry points should exist
- CA1401: P/Invokes should not be visible
- CA2101: Specify marshaling for P/Invoke string arguments

CA2207: Initialize value type static fields inline

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2207
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A value-type declares an explicit static constructor.

Rule description

When a value-type is declared, it undergoes a default initialization where all value-type fields are set to zero and all reference-type fields are set to `null` (`Nothing` in Visual Basic). An explicit static constructor is only guaranteed to run before an instance constructor or static member of the type is called. Therefore, if the type is created without calling an instance constructor, the static constructor is not guaranteed to run.

If all static data is initialized inline and no explicit static constructor is declared, the C# and Visual Basic compilers add the `beforefieldinit` flag to the MSIL class definition. The compilers also add a private static constructor that contains the static initialization code. This private static constructor is guaranteed to run before any static fields of the type are accessed.

How to fix violations

To fix a violation of this rule initialize all static data when it is declared and remove the static constructor.

When to suppress warnings

Do not suppress a warning from this rule.

Related rules

CA1810: Initialize reference type static fields inline

CA2208: Instantiate argument exceptions correctly

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2208
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

When a method has a parameter, and it throws an exception type that is, or derives from, ArgumentException, it expected to call a constructor accepting a `paramName` parameter correctly. Possible causes include the following situations:

- A call is made to the default (parameterless) constructor of an exception type that is, or derives from, ArgumentException that has a constructor accepting a `paramName` parameter.
- An incorrect string argument is passed to a parameterized constructor of an exception type that is, or derives from, ArgumentException.
- One of the parameters' names is passed for the `message` argument of the constructor of exception type that is, or derives from, ArgumentException.

Rule description

Instead of calling the default constructor, call one of the constructor overloads that allows a more meaningful exception message to be provided. The exception message should target the developer and clearly explain the error condition and how to correct or avoid the exception.

The signatures of the one and two string constructors of ArgumentException and its derived types are not consistent with respect to the position `message` and `paramName` parameters. Make sure these constructors are called with the correct string arguments. The signatures are as follows:

- ArgumentException(string `message`)
- ArgumentException(string `message`, string `paramName`)
- ArgumentNullException(string `paramName`)
- ArgumentNullException(string `paramName`, string `message`)
- ArgumentOutOfRangeException(string `paramName`)
- ArgumentOutOfRangeException(string `paramName`, string `message`)
- DuplicateWaitObjectException(string `parameterName`)
- DuplicateWaitObjectException(string `parameterName`, string `message`)

How to fix violations

To fix a violation of this rule, call a constructor that takes a message, a parameter name, or both, and make sure the arguments are proper for the type of ArgumentException being called.

TIP

A code fix is available for incorrectly positioned parameter name in Visual Studio. To use it, position the cursor on the warning row and press **Ctrl+.** (period). Choose **Swap the arguments order** from the list of options that's presented.

In case the ArgumentException(string `message`) overload is used mistakenly with a parameter name, the fixer would provide the option to use a two arguments constructor with `null` message.

When to suppress warnings

It's safe to suppress a warning from this rule only if a parameterized constructor is called with the correct string arguments.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca2208.api_surface = public
```

By default CA2208 rules apply for all API surface (public, internal and private). You can configure this option for just this rule, for all rules, or for all rules in this category (Design). For more information, see Configure FxCop analyzers.

Example

The following code shows a constructor that incorrectly instantiates an instance of ArgumentNullException.

```
using namespace System;

namespace Samples1
{
    public ref class Book
    {
     private: initonly String^ _Title;

    public:
        Book(String^ title)
        {
            // Violates this rule (constructor arguments are switched)
            if (title == nullptr)
                throw gcnew ArgumentNullException("title cannot be a null reference (Nothing in Visual
Basic)", "title");

            _Title = title;
        }

        property String^ Title
        {
            String^ get()
            {
                return _Title;
            }
        }
    };
}
```

```
public Book(string title)
{
    Title = title ?? throw new ArgumentNullException("All books must have a title.", nameof(title));
}
```

```
Imports System

Namespace Samples1

    Public Class Book

        Private ReadOnly _Title As String

        Public Sub New(ByVal title As String)
            ' Violates this rule (constructor arguments are switched)
            If (title Is Nothing) Then
                Throw New ArgumentNullException("title cannot be a null reference (Nothing in Visual Basic)",
"title")
            End If
            _Title = title
        End Sub

        Public ReadOnly Property Title()
            Get
                Return _Title
            End Get
        End Property

    End Class

End Namespace
```

The following code fixes the previous violation by switching the constructor arguments.

```
using namespace System;

namespace Samples2
{
    public ref class Book
    {
     private: initonly String^ _Title;

    public:
        Book(String^ title)
        {
            if (title == nullptr)
                throw gcnew ArgumentNullException(("title", "title cannot be a null reference (Nothing in
Visual Basic)"));

            _Title = title;
        }

        property String^ Title
        {
            String^ get()
            {
                return _Title;
            }
        }
    };
}
```

```
public Book(string title)
{
    Title = title ?? throw new ArgumentNullException(nameof(title), "All books must have a title.");
}
```

```
Namespace Samples2

    Public Class Book

        Private ReadOnly _Title As String

        Public Sub New(ByVal title As String)
            If (title Is Nothing) Then
                Throw New ArgumentNullException("title", "title cannot be a null reference (Nothing in Visual
Basic)")
            End If

            _Title = title
        End Sub

        Public ReadOnly Property Title()
            Get
                Return _Title
            End Get
        End Property

    End Class

End Namespace
```

Related rules

- CA1507: Use nameof in place of string

CA2211: Non-constant fields should not be visible

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2211
Category	Microsoft.Usage
Breaking change	Breaking

Cause

A public or protected static field is not constant nor is it read-only.

Rule description

Static fields that are neither constants nor read-only are not thread-safe. Access to such a field must be carefully controlled and requires advanced programming techniques for synchronizing access to the class object. Because these are difficult skills to learn and master, and testing such an object poses its own challenges, static fields are best used to store data that does not change. This rule applies to libraries; applications should not expose any fields.

How to fix violations

To fix a violation of this rule, make the static field constant or read-only. If this is not possible, redesign the type to use an alternative mechanism such as a thread-safe property that manages thread-safe access to the underlying field. Realize that issues such as lock contention and deadlocks might affect the performance and behavior of the library.

When to suppress warnings

It is safe to suppress a warning from this rule if you are developing an application and therefore have full control over access to the type that contains the static field. Library designers should not suppress a warning from this rule; using non-constant static fields can make using the library difficult for developers to use correctly.

Example

The following example shows a type that violates this rule.

```
Imports System

Namespace UsageLibrary

Public Class SomeStaticFields
    ' Violates rule: AvoidNonConstantStatic;
    ' the field is public and not a literal.
    Public Shared publicField As DateTime = DateTime.Now

    ' Satisfies rule: AvoidNonConstantStatic.
    Public Shared ReadOnly literalField As DateTime = DateTime.Now

    ' Satisfies rule: NonConstantFieldsShouldNotBeVisible;
    ' the field is private.
    Private Shared privateField As DateTime = DateTime.Now
End Class
End Namespace
```

```
using System;

namespace UsageLibrary
{
   public class SomeStaticFields
   {
      // Violates rule: AvoidNonConstantStatic;
      // the field is public and not a literal.
      static public DateTime publicField = DateTime.Now;

      // Satisfies rule: AvoidNonConstantStatic.
      public static readonly DateTime literalField = DateTime.Now;

      // Satisfies rule: NonConstantFieldsShouldNotBeVisible;
      // the field is private.
      static DateTime privateField = DateTime.Now;
   }
}
```

CA2212: Do not mark serviced components with WebMethod

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2212
Category	Microsoft.Usage
Breaking change	Breaking

Cause

A method in a type that inherits from System.EnterpriseServices.ServicedComponent is marked with System.Web.Services.WebMethodAttribute.

Rule description

WebMethodAttribute applies to methods within an XML web service that were created by using ASP.NET; it makes the method callable from remote web clients. The method and class must be public and executing in an ASP.NET web application. ServicedComponent types are hosted by COM+ applications and can use COM+ services. WebMethodAttribute is not applied to ServicedComponent types because they are not intended for the same scenarios. Specifically, adding the attribute to the ServicedComponent method does not make the method callable from remote web clients. Because WebMethodAttribute and a ServicedComponent method have conflicting behaviors and requirements for context and transaction flow, the behavior of the method will be incorrect in some scenarios.

How to fix violations

To fix a violation of this rule, remove the attribute from the ServicedComponent method.

When to suppress warnings

Do not suppress a warning from this rule. There are no scenarios where combining these elements is correct.

See also

- System.EnterpriseServices.ServicedComponent
- System.Web.Services.WebMethodAttribute

CA2213: Disposable fields should be disposed

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2213
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A type that implements System.IDisposable declares fields that are of types that also implement IDisposable. The Dispose method of the field is not called by the Dispose method of the declaring type.

Rule description

A type is responsible for disposing of all its unmanaged resources. Rule CA2213 checks to see whether a disposable type (that is, one that implements IDisposable) `T` declares a field `F` that is an instance of a disposable type `FT`. For each field `F` that's assigned a locally created object within the methods or initializers of the containing type `T`, the rule attempts to locate a call to `FT.Dispose`. The rule searches the methods called by `T.Dispose` and one level lower (that is, the methods called by the methods called by `T.Dispose`).

> **NOTE**
>
> Other than the special cases, rule CA2213 fires only for fields that are assigned a locally created disposable object within the containing type's methods and initializers. If the object is created or assigned outside of type `T`, the rule does not fire. This reduces noise for cases where the containing type doesn't own the responsibility for disposing of the object.

Special cases

Rule CA2213 can also fire for fields of the following types even if the object they're assigned isn't created locally:

- System.IO.Stream
- System.IO.TextReader
- System.IO.TextWriter
- System.Resources.IResourceReader

Passing an object of one of these types to a constructor and then assigning it to a field indicates a *dispose ownership transfer* to the newly constructed type. That is, the newly constructed type is now responsible for disposing of the object. If the object is not disposed, a violation of CA2213 occurs.

How to fix violations

To fix a violation of this rule, call Dispose on fields that are of types that implement IDisposable.

When to suppress warnings

It's safe to suppress a warning from this rule if:

- The flagged type is not responsible for releasing the resource held by the field (that is, the type does not have *dispose ownership*)
- The call to Dispose occurs at a deeper calling level than the rule checks
- the dispose ownership of the field(s) is not held by the containing type.

Example

The following snippet shows a type `TypeA` that implements IDisposable.

```
using System;

namespace UsageLibrary
{
    public class  TypeA : IDisposable
    {
        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                // Dispose managed resources
            }

            // Free native resources
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        // Disposable types implement a finalizer.
        ~TypeA()
        {
            Dispose(false);
        }
    }
}
```

The following snippet shows a type `TypeB` that violates rule CA2213 by declaring a field `aFieldOfADisposableType` as a disposable type (`TypeA`) and not calling Dispose on the field.

```
using System;

namespace UsageLibrary
{
   public class  TypeB : IDisposable
   {
      // Assume this type has some unmanaged resources.
      TypeA aFieldOfADisposableType = new TypeA();
      private bool disposed = false;

      protected virtual void Dispose(bool disposing)
      {
         if (!disposed)
         {
            // Dispose of resources held by this instance.

            // Violates rule: DisposableFieldsShouldBeDisposed.
            // Should call aFieldOfADisposableType.Dispose();

            disposed = true;
            // Suppress finalization of this disposed instance.
            if (disposing)
            {
                GC.SuppressFinalize(this);
            }
         }
      }

      public void Dispose()
      {
         if (!disposed)
         {
            // Dispose of resources held by this instance.
            Dispose(true);
         }
      }

      // Disposable types implement a finalizer.
      ~TypeB()
      {
         Dispose(false);
      }
   }
}
```

To fix the violation, call `Dispose()` on the disposable field:

```
protected virtual void Dispose(bool disposing)
{
   if (!disposed)
   {
      // Dispose of resources held by this instance.
      aFieldOfADisposableType.Dispose();

      disposed = true;

      // Suppress finalization of this disposed instance.
      if (disposing)
      {
          GC.SuppressFinalize(this);
      }
   }
}
```

See also

- System.IDisposable
- Dispose pattern

CA2214: Do not call overridable methods in constructors

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2214
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

The constructor of an unsealed type calls a virtual method defined in its class.

Rule description

When a virtual method is called, the actual type that executes the method is not selected until run time. When a constructor calls a virtual method, it's possible that the constructor for the instance that invokes the method has not executed. This could lead to errors or unexpected behavior, if an overridden virtual method relies on initialization and other configuration in the constructor.

> **NOTE**
>
> The legacy analysis implementation of this rule has a different diagnostic message of "**[Constructor name] contains a call chain that results in a call to a virtual method defined by the class. Review the following call stack for unintended consequences**". The FxCop analyzers implementation of this rule has a diagnostic message of "**Do not call overridable methods in constructors**".

How to fix violations

To fix a violation of this rule, do not call a type's virtual methods from within the type's constructors.

When to suppress warnings

Do not suppress a warning from this rule. The constructor should be redesigned to eliminate the call to the virtual method.

Example

The following example demonstrates the effect of violating this rule. The test application creates an instance of `DerivedType`, which causes its base class (`BadlyConstructedType`) constructor to execute. `BadlyConstructedType`'s constructor incorrectly calls the virtual method `DoSomething`. As the output shows, `DerivedType.DoSomething()` executes before `DerivedType`'s constructor executes.

```
using System;

namespace UsageLibrary
{
    public class BadlyConstructedType
    {
        protected  string initialized = "No";

        public BadlyConstructedType()
        {
            Console.WriteLine("Calling base ctor.");
            // Violates rule: DoNotCallOverridableMethodsInConstructors.
            DoSomething();
        }
        // This will be overridden in the derived type.
        public virtual void DoSomething()
        {
            Console.WriteLine ("Base DoSomething");
        }
    }

    public class DerivedType : BadlyConstructedType
    {
        public DerivedType ()
        {
            Console.WriteLine("Calling derived ctor.");
            initialized = "Yes";
        }
        public override void DoSomething()
        {
            Console.WriteLine("Derived DoSomething is called - initialized ? {0}", initialized);
        }
    }

    public class TestBadlyConstructedType
    {
        public static void Main()
        {
            DerivedType derivedInstance = new DerivedType();
        }
    }
}
```

```
Imports System

Namespace UsageLibrary

Public Class BadlyConstructedType
    Protected initialized As String = "No"


    Public Sub New()
        Console.WriteLine("Calling base ctor.")
        ' Violates rule: DoNotCallOverridableMethodsInConstructors.
        DoSomething()
    End Sub 'New

    ' This will be overridden in the derived type.
    Public Overridable Sub DoSomething()
        Console.WriteLine("Base DoSomething")
    End Sub 'DoSomething
End Class 'BadlyConstructedType


Public Class DerivedType
    Inherits BadlyConstructedType

    Public Sub New()
        Console.WriteLine("Calling derived ctor.")
        initialized = "Yes"
    End Sub 'New

    Public Overrides Sub DoSomething()
        Console.WriteLine("Derived DoSomething is called - initialized ? {0}", initialized)
    End Sub 'DoSomething
End Class 'DerivedType


Public Class TestBadlyConstructedType

    Public Shared Sub Main()
        Dim derivedInstance As New DerivedType()
    End Sub 'Main
End Class
End Namespace
```

This example produces the following output:

```
Calling base ctor.
Derived DoSomething is called - initialized ? No
Calling derived ctor.
```

CA2215: Dispose methods should call base class dispose

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2215
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A type that implements System.IDisposable inherits from a type that also implements IDisposable. The Dispose method of the inheriting type does not call the Dispose method of the parent type.

Rule description

If a type inherits from a disposable type, it must call the Dispose method of the base type from within its own Dispose method. Calling the base type method Dispose ensures that any resources created by the base type are released.

How to fix violations

To fix a violation of this rule, call `base`.Dispose in your Dispose method.

When to suppress warnings

It is safe to suppress a warning from this rule if the call to `base`.Dispose occurs at a deeper calling level than the rule checks.

Example

The following example shows a type `TypeA` that implements IDisposable.

```
using System;

namespace UsageLibrary
{
    public class  TypeA :IDisposable
    {

        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                // Dispose managed resources
            }

            // Free native resources
        }

        public void Dispose()
        {

                Dispose(true);

                GC.SuppressFinalize(this);

        }

        // Disposable types implement a finalizer.
        ~TypeA()
        {
            Dispose(false);
        }
    }
}
```

Example

The following example shows a type `TypeB` that inherits from type `TypeA` and correctly calls its Dispose method.

```
Imports System

Namespace UsageLibrary

  Public Class TypeB
      Inherits TypeA

      Protected Overrides Sub Finalize()
          Try
              Dispose(False)
          Finally
              MyBase.Finalize()
          End Try
      End Sub

  End Class

End Namespace
```

See also

- System.IDisposable
- Dispose Pattern

CA2216: Disposable types should declare finalizer

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2216
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A type that implements System.IDisposable, and has fields that suggest the use of unmanaged resources, does not implement a finalizer as described by System.Object.Finalize.

Rule description

A violation of this rule is reported if the disposable type contains fields of the following types:

- System.IntPtr
- System.UIntPtr
- System.Runtime.InteropServices.HandleRef

How to fix violations

To fix a violation of this rule, implement a finalizer that calls your Dispose method.

When to suppress warnings

It is safe to suppress a warning from this rule if the type does not implement IDisposable for the purpose of releasing unmanaged resources.

Example

The following example shows a type that violates this rule.

```
using System;
using System.Runtime.InteropServices;

namespace UsageLibrary
{
    public class  DisposeMissingFinalize :IDisposable
    {
        private bool disposed = false;
        private IntPtr unmanagedResource;

        [DllImport("native.dll")]
        private static extern IntPtr AllocateUnmanagedResource();

        [DllImport("native.dll")]
        private static extern void FreeUnmanagedResource(IntPtr p);

        DisposeMissingFinalize()
        {
            unmanagedResource = AllocateUnmanagedResource();
        }

        protected virtual void Dispose(bool disposing)
        {
            if (!disposed)
            {
                // Dispose of resources held by this instance.
                FreeUnmanagedResource(unmanagedResource);
                disposed = true;

                // Suppress finalization of this disposed instance.
                if (disposing)
                {
                    GC.SuppressFinalize(this);
                }
            }
        }

        public void Dispose()
        {
            Dispose(true);
        }

        // Disposable types with unmanaged resources implement a finalizer.
        // Uncomment the following code to satisfy rule:
        //  DisposableTypesShouldDeclareFinalizer
        // ~TypeA()
        // {
        //     Dispose(false);
        // }
    }
}
```

Related rules

CA2115: Call GC.KeepAlive when using native resources

CA1816: Call GC.SuppressFinalize correctly

CA1049: Types that own native resources should be disposable

See also

- System.IDisposable
- System.IntPtr

- System.Runtime.InteropServices.HandleRef
- System.UIntPtr
- System.Object.Finalize
- Dispose Pattern

CA2217: Do not mark enums with FlagsAttribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2217
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An enumeration is marked with FlagsAttribute and it has one or more values that are not powers of two or a combination of the other defined values on the enumeration.

By default, this rule only looks at externally visible enumerations, but this is configurable.

Rule description

An enumeration should have FlagsAttribute present only if each value defined in the enumeration is a power of two or a combination of defined values.

How to fix violations

To fix a violation of this rule, remove FlagsAttribute from the enumeration.

When to suppress warnings

Do not suppress a warning from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca2217.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Usage). For more information, see Configure FxCop analyzers.

Examples

The following code shows an enumeration, `Color`, that contains the value 3. 3 is not a power of two, or a combination of any of the defined values. The `Color` enumeration shouldn't be marked with FlagsAttribute.

```
using namespace System;

namespace Samples
{
    // Violates this rule
    [FlagsAttribute]
    public enum class Color
    {
        None   = 0,
        Red    = 1,
        Orange = 3,
        Yellow = 4
    };
}
```

```
using System;

namespace Samples
{
    // Violates this rule
    [FlagsAttribute]
    public enum Color
    {
        None    = 0,
        Red     = 1,
        Orange  = 3,
        Yellow  = 4
    }
}
```

```
Imports System

Namespace Samples

    ' Violates this rule
    <FlagsAttribute()> _
    Public Enum Color

        None = 0
        Red = 1
        Orange = 3
        Yellow = 4

    End Enum
End Namespace
```

The following code shows an enumeration, `Days`, that meets the requirements for being marked with FlagsAttribute:

```
using namespace System;

namespace Samples
{
    [FlagsAttribute]
    public enum class Days
    {
        None        = 0,
        Monday      = 1,
        Tuesday     = 2,
        Wednesday   = 4,
        Thursday    = 8,
        Friday      = 16,
        All         = Monday| Tuesday | Wednesday | Thursday | Friday
    };
}
```

```
using System;

namespace Samples
{
    [FlagsAttribute]
    public enum Days
    {
        None        = 0,
        Monday      = 1,
        Tuesday     = 2,
        Wednesday   = 4,
        Thursday    = 8,
        Friday      = 16,
        All         = Monday| Tuesday | Wednesday | Thursday | Friday
    }
}
```

```
Imports System
Namespace Samples

    <FlagsAttribute()> _
    Public Enum Days

        None = 0
        Monday = 1
        Tuesday = 2
        Wednesday = 4
        Thursday = 8
        Friday = 16
        All = Monday Or Tuesday Or Wednesday Or Thursday Or Friday

    End Enum
End Namespace
```

Related rules

CA1027: Mark enums with FlagsAttribute

See also

- System.FlagsAttribute

CA2218: Override GetHashCode on overriding Equals

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2218
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A public type overrides System.Object.Equals but does not override System.Object.GetHashCode.

Rule description

GetHashCode returns a value, based on the current instance, that is suited for hashing algorithms and data structures such as a hash table. Two objects that are the same type and are equal must return the same hash code to ensure that instances of the following types work correctly:

- System.Collections.Hashtable
- System.Collections.SortedList
- System.Collections.Generic.Dictionary<TKey,TValue>
- System.Collections.Generic.SortedDictionary<TKey,TValue>
- System.Collections.Generic.SortedList<TKey,TValue>
- System.Collections.Specialized.HybridDictionary
- System.Collections.Specialized.ListDictionary
- System.Collections.Specialized.OrderedDictionary
- Types that implement System.Collections.Generic.IEqualityComparer<T>

How to fix violations

To fix a violation of this rule, provide an implementation of GetHashCode. For a pair of objects of the same type, you must ensure that the implementation returns the same value if your implementation of Equals returns `true` for the pair.

When to suppress warnings

Do not suppress a warning from this rule.

Class Example

Description

The following example shows a class (reference type) that violates this rule.

Code

```
using System;

namespace Samples
{
    // Violates this rule
    public class Point
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override bool Equals(object obj)
        {
            if (obj == null)
                return false;

            if (GetType() != obj.GetType())
                return false;

            Point point = (Point)obj;

            if (_X != point.X)
                return false;

            return _Y == point.Y;
        }
    }
}
```

Comments

The following example fixes the violation by overriding GetHashCode().

Code

```
using System;

namespace Samples
{
    public struct Point : IEquatable<Point>
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override int GetHashCode()
        {
            return _X ^ _Y;
        }

        public override bool Equals(object obj)
        {
            if (!(obj is Point))
                return false;

            return Equals((Point)obj);
        }

        public bool Equals(Point other)
        {
            if (_X != other._X)
                return false;

            return _Y == other._Y;
        }

        public static bool operator ==(Point point1, Point point2)
        {
            return point1.Equals(point2);
        }

        public static bool operator !=(Point point1, Point point2)
        {
            return !point1.Equals(point2);
        }
    }
}
```

Structure Example

Description

The following example shows a structure (value type) that violates this rule.

Code

```
using System;

namespace Samples
{
    // Violates this rule
    public struct Point : IEquatable<Point>
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override bool Equals(object obj)
        {
            if (!(obj is Point))
                return false;

            return Equals((Point)obj);
        }

        public bool Equals(Point other)
        {
            if (_X != other._X)
                return false;

            return _Y == other._Y;
        }

        public static bool operator ==(Point point1, Point point2)
        {
            return point1.Equals(point2);
        }

        public static bool operator !=(Point point1, Point point2)
        {
            return !point1.Equals(point2);
        }
    }
}
```

Comments

The following example fixes the violation by overriding GetHashCode().

Code

```
using System;

namespace Samples
{
    public struct Point : IEquatable<Point>
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override int GetHashCode()
        {
            return _X ^ _Y;
        }

        public override bool Equals(object obj)
        {
            if (!(obj is Point))
                return false;

            return Equals((Point)obj);
        }

        public bool Equals(Point other)
        {
            if (_X != other._X)
                return false;

            return _Y == other._Y;
        }

        public static bool operator ==(Point point1, Point point2)
        {
            return point1.Equals(point2);
        }

        public static bool operator !=(Point point1, Point point2)
        {
            return !point1.Equals(point2);
        }
    }
}
```

Related rules

CA1046: Do not overload operator equals on reference types

CA2225: Operator overloads have named alternates

CA2226: Operators should have symmetrical overloads

CA2224: Override equals on overloading operator equals

CA2231: Overload operator equals on overriding ValueType.Equals

See also

- System.Object.Equals
- System.Object.GetHashCode
- System.Collections.Hashtable
- Equality Operators

CA2219: Do not raise exceptions in exception clauses

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2219
Category	Microsoft.Usage
Breaking change	Non-breaking, Breaking

Cause

An exception is thrown from a `finally`, filter, or fault clause.

Rule description

When an exception is raised in an exception clause, it greatly increases the difficulty of debugging.

When an exception is raised in a `finally` or fault clause, the new exception hides the active exception, if present. This makes the original error hard to detect and debug.

When an exception is raised in a filter clause, the runtime silently catches the exception, and causes the filter to evaluate to false. There is no way to tell the difference between the filter evaluating to false and an exception being throw from a filter. This makes it hard to detect and debug errors in the filter's logic.

How to fix violations

To fix this violation of this rule, do not explicitly raise an exception from a `finally`, filter, or fault clause.

When to suppress warnings

Do not suppress a warning for this rule. There are no scenarios under which an exception raised in an exception clause provides a benefit to the executing code.

Related rules

CA1065: Do not raise exceptions in unexpected locations

See also

Design Warnings

CA2220: Finalizers should call base class finalizer

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2220
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A type that overrides System.Object.Finalize does not call the Finalize method in its base class.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Finalization must be propagated through the inheritance hierarchy. To ensure this, types must call their base class Finalize method from within their own Finalize method. The C# compiler adds the call to the base class finalizer automatically.

How to fix violations

To fix a violation of this rule, call the base type's Finalize method from your Finalize method.

When to suppress warnings

Do not suppress a warning from this rule. Some compilers that target the common language runtime insert a call to the base type's finalizer into the Microsoft intermediate language (MSIL). If a warning from this rule is reported, your compiler does not insert the call, and you must add it to your code.

Example

The following Visual Basic example shows a type `TypeB` that correctly calls the Finalize method in its base class.

```
Imports System

Namespace UsageLibrary

  Public Class TypeB
      Inherits TypeA

      Protected Overrides Sub Finalize()
          Try
              Dispose(False)
          Finally
              MyBase.Finalize()
          End Try
      End Sub

  End Class

End Namespace
```

See also

- Dispose Pattern

CA2221: Finalizers should be protected

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2221
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A public type implements a finalizer that does not specify family (protected) access.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Finalizers must use the family access modifier. This rule is enforced by the C#, Visual Basic, and C++ compilers.

How to fix violations

To fix a violation of this rule, change the finalizer to be family-accessible.

When to suppress warnings

Do not suppress a warning from this rule.

Example

This rule cannot be violated in any high-level .NET language; it can be violated if you are writing Microsoft Intermediate Language.

```
// =============== CLASS MEMBERS DECLARATION ===================
//   note that class flags, 'extends' and 'implements' clauses
//          are provided here for information only

.namespace UsageLibrary
{
  .class public auto ansi beforefieldinit FinalizeMethodNotProtected
         extends [mscorlib]System.Object
  {
    .method public hidebysig instance void
            Finalize() cil managed
    {

      // Code size       1 (0x1)
      .maxstack  0
      IL_0000:  ret
    } // end of method FinalizeMethodNotProtected::Finalize

    .method public hidebysig specialname rtspecialname
            instance void  .ctor() cil managed
    {
      // Code size       7 (0x7)
      .maxstack  1
      IL_0000:  ldarg.0
      IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
      IL_0006:  ret
    } // end of method FinalizeMethodNotProtected::.ctor

  } // end of class FinalizeMethodNotProtected
} // end of namespace
```

See also

- Dispose Pattern

CA2222: Do not decrease inherited member visibility

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2222
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A private method in an unsealed type has a signature that is identical to a public method declared in a base type. The private method is not final.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Don't change the access modifier for inherited members. Changing an inherited member to private does not prevent callers from accessing the base class implementation of the method. If the member is made private and the type is unsealed, inheriting types can call the last public implementation of the method in the inheritance hierarchy. If you must change the access modifier, either the method should be marked final or its type should be sealed to prevent the method from being overridden.

How to fix violations

To fix a violation of this rule, change the access to be non-private. Alternatively, if your programming language supports it, you can make the method final.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a type that violates this rule.

```
Imports System

Namespace UsageLibrary
Public Class ABaseType

   Public Sub BasePublicMethod(argument1 As Integer)
   End Sub 'BasePublicMethod

End Class 'ABaseType

Public Class ADerivedType
   Inherits ABaseType

   ' Violates rule DoNotDecreaseInheritedMemberVisibility.
   Private Shadows Sub BasePublicMethod(argument1 As Integer)
   End Sub 'BasePublicMethod
End Class 'ADerivedType

End Namespace
```

```
using System;
namespace UsageLibrary
{
    public class ABaseType
    {
        public void BasePublicMethod(int argument1) {}
    }
    public class ADerivedType:ABaseType
    {
        // Violates rule: DoNotDecreaseInheritedMemberVisibility.
        // The compiler returns an error if this is overridden instead of new.
        private new void BasePublicMethod(int argument1){}
    }
}
```

CA2223: Members should differ by more than return type

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2223
Category	Microsoft.Usage
Breaking change	Breaking

Cause

Two public or protected members have signatures that are identical except for return type.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Although the common language runtime permits the use of return types to differentiate between otherwise identical members, this feature is not in the Common Language Specification, nor is it a common feature of .NET programming languages. When members differ only by return type, developers and development tools might not correctly distinguish between them.

How to fix violations

To fix a violation of this rule, change the design of the members so that they are unique based only on their names and parameter types, or do not expose the members.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example, in Microsoft intermediate language (MSIL), shows a type that violates this rule. Notice that this rule cannot be violated by using C# or Visual Basic.

```
.namespace UsageLibrary
{
  .class public auto ansi beforefieldinit ReturnTypeTest
          extends [mscorlib]System.Object
  {
    .method public hidebysig instance int32
            AMethod(int32 x) cil managed
    {
      // Code size       6 (0x6)
      .maxstack  1
      .locals init (int32 V_0)
      IL_0000:  ldc.i4.0
      IL_0001:  stloc.0
      IL_0002:  br.s       IL_0004

      IL_0004:  ldloc.0
      IL_0005:  ret
    } // end of method ReturnTypeTest::AMethod

    .method public hidebysig instance string
            AMethod(int32 x) cil managed
    {
      // Code size       10 (0xa)
      .maxstack  1
      .locals init (string V_0)
      IL_0000:  ldstr      "test"
      IL_0005:  stloc.0
      IL_0006:  br.s       IL_0008

      IL_0008:  ldloc.0
      IL_0009:  ret
    } // end of method ReturnTypeTest::AMethod

    .method public hidebysig specialname rtspecialname
            instance void  .ctor() cil managed
    {
      // Code size       7 (0x7)
      .maxstack  1
      IL_0000:  ldarg.0
      IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
      IL_0006:  ret
    } // end of method ReturnTypeTest::.ctor

  } // end of class ReturnTypeTest

} // end of namespace UsageLibrary
```

CA2224: Override equals on overloading operator equals

8/4/2020 • 6 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2224
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A public type implements the equality operator, but does not override System.Object.Equals.

Rule description

The equality operator is intended to be a syntactically convenient way to access the functionality of the Equals method. If you implement the equality operator, its logic must be identical to that of Equals.

The C# compiler issues a warning if your code violates this rule.

How to fix violations

To fix a violation of this rule, you should either remove the implementation of the equality operator, or override Equals and have the two methods return the same values. If the equality operator does not introduce inconsistent behavior, you can fix the violation by providing an implementation of Equals that calls the Equals method in the base class.

When to suppress warnings

It is safe to suppress a warning from this rule if the equality operator returns the same value as the inherited implementation of Equals. The examples in this article include a type that could safely suppress a warning from this rule.

Examples of Inconsistent Equality Definitions

The following example shows a type with inconsistent definitions of equality. `BadPoint` changes the meaning of equality by providing a custom implementation of the equality operator, but does not override Equals so that it behaves identically.

```
using System;

namespace UsageLibrary
{
    public class BadPoint
    {
        private int x,y, id;
        private static int NextId;

        static BadPoint()
        {
            NextId = -1;
        }
        public BadPoint(int x, int y)
        {
            this.x = x;
            this.y = y;
            id = ++(BadPoint.NextId);
        }

        public override string ToString()
        {
            return String.Format("([{0}] {1},{2})",id,x,y);
        }

        public int X {get {return x;}}

        public int Y {get {return x;}}
        public int Id {get {return id;}}

        public override int GetHashCode()
        {
            return id;
        }
        // Violates rule: OverrideEqualsOnOverridingOperatorEquals.

        // BadPoint redefines the equality operator to ignore the id value.
        // This is different from how the inherited implementation of
        // System.Object.Equals behaves for value types.
        // It is not safe to exclude the violation for this type.
        public static bool operator== (BadPoint p1, BadPoint p2)
        {
            return ((p1.x == p2.x) && (p1.y == p2.y));
        }
        // The C# compiler and rule OperatorsShouldHaveSymmetricalOverloads require this.
        public static bool operator!= (BadPoint p1, BadPoint p2)
        {
            return !(p1 == p2);
        }
    }
}
```

The following code tests the behavior of `BadPoint`.

```
using System;

namespace UsageLibrary
{
    public class TestBadPoint
    {
        public static void Main()
        {
            BadPoint a = new BadPoint(1,1);
            BadPoint b = new BadPoint(2,2);
            BadPoint a1 = a;
            BadPoint bcopy = new BadPoint(2,2);

            Console.WriteLine("a =  {0} and b = {1} are equal? {2}", a, b, a.Equals(b)? "Yes":"No");
            Console.WriteLine("a == b ? {0}", a == b ? "Yes":"No");
            Console.WriteLine("a1 and a are equal? {0}", a1.Equals(a)? "Yes":"No");
            Console.WriteLine("a1 == a ? {0}", a1 == a ? "Yes":"No");

            // This test demonstrates the inconsistent behavior of == and Object.Equals.
            Console.WriteLine("b and bcopy are equal ? {0}", bcopy.Equals(b)? "Yes":"No");
            Console.WriteLine("b == bcopy ? {0}", b == bcopy ? "Yes":"No");
        }
    }
}
```

This example produces the following output:

```
a =  ([0] 1,1) and b = ([1] 2,2) are equal? No
a == b ? No
a1 and a are equal? Yes
a1 == a ? Yes
b and bcopy are equal ? No
b == bcopy ? Yes
```

The following example shows a type that technically violates this rule, but does not behave in an inconsistent manner.

```
using System;

namespace UsageLibrary
{
    public struct GoodPoint
    {
        private int x,y;

        public GoodPoint(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

        public override string ToString()
        {
            return String.Format("({0},{1})",x,y);
        }

        public int X {get {return x;}}

        public int Y {get {return x;}}

        // Violates rule: OverrideEqualsOnOverridingOperatorEquals,
        // but does not change the meaning of equality;
        //  the violation can be excluded.

        public static bool operator== (GoodPoint px, GoodPoint py)
        {
            return px.Equals(py);
        }

        // The C# compiler and rule OperatorsShouldHaveSymmetricalOverloads require this.
        public static bool operator!= (GoodPoint px, GoodPoint py)
        {
            return !(px.Equals(py));
        }
    }
}
```

The following code tests the behavior of `GoodPoint`.

```
using System;

namespace UsageLibrary
{
    public class TestGoodPoint
    {
        public static void Main()
        {
            GoodPoint a = new GoodPoint(1,1);
            GoodPoint b = new GoodPoint(2,2);
            GoodPoint a1 = a;
            GoodPoint bcopy = new GoodPoint(2,2);

            Console.WriteLine("a =  {0} and b = {1} are equal? {2}", a, b, a.Equals(b)? "Yes":"No");
            Console.WriteLine("a == b ? {0}", a == b ? "Yes":"No");
            Console.WriteLine("a1 and a are equal? {0}", a1.Equals(a)? "Yes":"No");
            Console.WriteLine("a1 == a ? {0}", a1 == a ? "Yes":"No");

            // This test demonstrates the consistent behavior of == and Object.Equals.
            Console.WriteLine("b and bcopy are equal ? {0}", bcopy.Equals(b)? "Yes":"No");
            Console.WriteLine("b == bcopy ? {0}", b == bcopy ? "Yes":"No");
        }
    }
}
```

This example produces the following output:

```
a =  (1,1) and b = (2,2) are equal? No
a == b ? No
a1 and a are equal? Yes
a1 == a ? Yes
b and bcopy are equal ? Yes
b == bcopy ? Yes
```

Class Example

The following example shows a class (reference type) that violates this rule.

```
using System;

namespace Samples
{
    // Violates this rule
    public class Point
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override int GetHashCode()
        {
            return _X ^ _Y;
        }

        public static bool operator ==(Point point1, Point point2)
        {
            if (point1 == null || point2 == null)
                return false;

            if (point1.GetType() != point2.GetType())
                return false;

            if (point1._X != point2._X)
                return false;

            return point1._Y == point2._Y;
        }

        public static bool operator !=(Point point1, Point point2)
        {
            return !(point1 == point2);
        }
    }
}
```

The following example fixes the violation by overriding System.Object.Equals.

```
using System;

namespace Samples
{
    public class Point
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override int GetHashCode()
        {
            return _X ^ _Y;
        }

        public override bool Equals(object obj)
        {
            if (obj == null)
                return false;

            if (GetType() != obj.GetType())
                return false;

            Point point = (Point)obj;

            if (_X != point.X)
                return false;

            return _Y == point.Y;
        }

        public static bool operator ==(Point point1, Point point2)
        {
            return Object.Equals(point1, point2);
        }

        public static bool operator !=(Point point1, Point point2)
        {
            return !Object.Equals(point1, point2);
        }
    }
}
```

Structure Example

The following example shows a structure (value type) that violates this rule:

```
using System;

namespace Samples
{
    // Violates this rule
    public struct Point
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override int GetHashCode()
        {
            return _X ^ _Y;
        }

        public static bool operator ==(Point point1, Point point2)
        {
            if (point1._X != point2._X)
                return false;

            return point1._Y == point2._Y;
        }

        public static bool operator !=(Point point1, Point point2)
        {
            return !(point1 == point2);
        }
    }
}
```

The following example fixes the violation by overriding System.ValueType.Equals.

```
using System;

namespace Samples
{
    public struct Point : IEquatable<Point>
    {
        private readonly int _X;
        private readonly int _Y;

        public Point(int x, int y)
        {
            _X = x;
            _Y = y;
        }

        public int X
        {
            get { return _X; }
        }

        public int Y
        {
            get { return _Y; }
        }

        public override int GetHashCode()
        {
            return _X ^ _Y;
        }

        public override bool Equals(object obj)
        {
            if (!(obj is Point))
                return false;

            return Equals((Point)obj);
        }

        public bool Equals(Point other)
        {
            if (_X != other._X)
                return false;

            return _Y == other._Y;
        }

        public static bool operator ==(Point point1, Point point2)
        {
            return point1.Equals(point2);
        }

        public static bool operator !=(Point point1, Point point2)
        {
            return !point1.Equals(point2);
        }
    }
}
```

Related rules

CA1046: Do not overload operator equals on reference types

CA2225: Operator overloads have named alternates

CA2226: Operators should have symmetrical overloads

CA2218: Override GetHashCode on overriding Equals

CA2231: Overload operator equals on overriding ValueType.Equals

CA2225: Operator overloads have named alternates

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2225
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An operator overload was detected and the expected named alternative method was not found.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

Operator overloading allows the use of symbols to represent computations for a type. For example, a type that overloads the plus symbol `+` for addition would typically have an alternative member named `Add`. The named alternative member provides access to the same functionality as the operator. It's provided for developers who program in languages that do not support overloaded operators.

This rule examines:

- Implicit and explicit cast operators in a type by checking for methods named `To<typename>` and `From<typename>`.
- The operators listed in the following table:

C#	VISUAL BASIC	C++	ALTERNATE METHOD NAME
+ (binary)	+	+ (binary)	Add
+=	+=	+=	Add
&	And	&	BitwiseAnd
&=	And=	&=	BitwiseAnd
|	Or	|	BitwiseOr
|=	Or=	|=	BitwiseOr
--	N/A	--	Decrement
/	/	/	Divide
/=	/=	/=	Divide

C#	VISUAL BASIC	C++	ALTERNATE METHOD NAME
==	=	==	Equals
^	Xor	^	Xor
^=	Xor=	^=	Xor
>	>	>	CompareTo or Compare
>=	>=	>=	CompareTo or Compare
++	N/A	++	Increment
!=	<>	!=	Equals
<<	<<	<<	LeftShift
<<=	<<=	<<=	LeftShift
<	<	<	CompareTo or Compare
<=	<=	<=	CompareTo or Compare
&&	N/A	&&	LogicalAnd
||	N/A	||	LogicalOr
!	N/A	!	LogicalNot
%	Mod	%	Mod or Remainder
%=	N/A	%=	Mod
* (binary)	*	*	Multiply
*=	N/A	*=	Multiply
~	Not	~	OnesComplement
>>	>>	>>	RightShift
=	N/A	>>=	RightShift
- (binary)	- (binary)	- (binary)	Subtract
-=	N/A	-=	Subtract
true	IsTrue	N/A	IsTrue (Property)
- (unary)	N/A	-	Negate

C#	VISUAL BASIC	C++	ALTERNATE METHOD NAME
+ (unary)	N/A	+	Plus
false	IsFalse	False	IsTrue (Property)

*N/A means the operator cannot be overloaded in the selected language.

> **NOTE**
> In C#, when a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded.

How to fix violations

To fix a violation of this rule, implement the alternative method for the operator. Name it using the recommended alternative name.

When to suppress warnings

Do not suppress a warning from this rule if you're implementing a shared library. Applications can ignore a warning from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca2225.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Usage). For more information, see Configure FxCop analyzers.

Example

The following example defines a structure that violates this rule. To correct the example, add a public `Add(int x, int y)` method to the structure.

```
using System;

namespace UsageLibrary
{
    public struct Point
    {
        private int x,y;

        public Point(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

        public override string ToString()
        {
            return String.Format("({0},{1})",x,y);
        }

        // Violates rule: OperatorOverloadsHaveNamedAlternates.
        public static Point operator+(Point a, Point b)
        {
            return new Point(a.x + b.x, a.y + b.y);
        }

        public int X {get {return x;}}
        public int Y {get {return x;}}
    }
}
```

Related rules

- CA1046: Do not overload operator equals on reference types
- CA2226: Operators should have symmetrical overloads
- CA2224: Override equals on overloading operator equals
- CA2218: Override GetHashCode on overriding Equals
- CA2231: Overload operator equals on overriding ValueType.Equals

CA2226: Operators should have symmetrical overloads

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2226
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A type implements the equality or inequality operator and does not implement the opposite operator.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

There are no circumstances where either equality or inequality is applicable to instances of a type, and the opposite operator is undefined. Types typically implement the inequality operator by returning the negated value of the equality operator.

The C# compiler issues an error for violations of this rule.

How to fix violations

To fix a violation of this rule, implement both the equality and inequality operators, or remove the one that's present.

When to suppress warnings

Do not suppress a warning from this rule. If you do, your type will not work in a manner that's consistent with .NET.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca2226.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Usage). For more information, see Configure FxCop analyzers.

Related rules

- CA1046: Do not overload operator equals on reference types
- CA2225: Operator overloads have named alternates
- CA2224: Override equals on overloading operator equals
- CA2218: Override GetHashCode on overriding Equals
- CA2231: Overload operator equals on overriding ValueType.Equals

CA2227: Collection properties should be read only

8/4/2020 • 3 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2227
Category	Microsoft.Usage
Breaking change	Breaking

Cause

An externally visible, writable property is of a type that implements System.Collections.ICollection. This rule ignores arrays, indexers (properties with the name 'Item'), immutable collections, readonly collections, and permission sets.

Rule description

A writable collection property allows a user to replace the collection with a completely different collection. A read-only property stops the collection from being replaced, but still allows the individual members to be set. If replacing the collection is a goal, the preferred design pattern is to include a method to remove all the elements from the collection, and a method to repopulate the collection. See the Clear and AddRange methods of the System.Collections.ArrayList class for an example of this pattern.

Both binary and XML serialization support read-only properties that are collections. The System.Xml.Serialization.XmlSerializer class has specific requirements for types that implement ICollection and System.Collections.IEnumerable in order to be serializable.

How to fix violations

To fix a violation of this rule, make the property read-only. If the design requires it, add methods to clear and repopulate the collection.

When to suppress warnings

You can suppress the warning if the property is part of a Data Transfer Object (DTO) class.

Otherwise, do not suppress warnings from this rule.

Example

The following example shows a type with a writable collection property and shows how the collection can be replaced directly. Additionally, it shows the preferred manner of replacing a read-only collection property using `Clear` and `AddRange` methods.

```
using System.Collections;

namespace csharp_code_analysis_examples
{
    public class WritableCollection
    {
        public ArrayList SomeStrings
        {
            get;

            // This set accessor violates rule CA2227.
            // To fix the code, remove this set accessor.
            set;
        }

        public WritableCollection()
        {
            SomeStrings = new ArrayList(new string[] { "one", "two", "three" });
        }
    }

    class ReplaceWritableCollection
    {
        static void Main()
        {
            ArrayList newCollection = new ArrayList(new string[] { "a", "new", "collection" });

            WritableCollection collection = new WritableCollection();

            // This line of code demonstrates how the entire collection
            // can be replaced by a property that's not read only.
            collection.SomeStrings = newCollection;

            // If the intent is to replace an entire collection,
            // implement and/or use the Clear() and AddRange() methods instead.
            collection.SomeStrings.Clear();
            collection.SomeStrings.AddRange(newCollection);
        }
    }
}
```

```
Public Class WritableCollection

    ' This property violates rule CA2227.
    ' To fix the code, add the ReadOnly modifier to the property:
    ' ReadOnly Property SomeStrings As ArrayList
    Property SomeStrings As ArrayList

    Sub New()
        SomeStrings = New ArrayList(New String() {"one", "two", "three"})
    End Sub

End Class

Class ViolatingVersusPreferred

    Shared Sub Main()
        Dim newCollection As New ArrayList(New String() {"a", "new", "collection"})

        Dim collection As New WritableCollection()

        ' This line of code demonstrates how the entire collection
        ' can be replaced by a property that's not read only.
        collection.SomeStrings = newCollection

        ' If the intent is to replace an entire collection,
        ' implement and/or use the Clear() and AddRange() methods instead.
        collection.SomeStrings.Clear()
        collection.SomeStrings.AddRange(newCollection)
    End Sub

End Class
```

```
#include "stdafx.h"
using namespace System;
using namespace System::Collections;

namespace UsageLibrary
{
    public ref class WritableCollection
    {
    public:
        property ArrayList^ SomeStrings
        {
            ArrayList^ get() { return someStrings; }

            // This set accessor violates rule CA2227.
            // To fix the code, remove this set accessor.
            void set(ArrayList^ value) { someStrings = value; }
        }

        WritableCollection()
        {
            someStrings = gcnew ArrayList(gcnew array<String^> {"one", "two", "three"});
        }

    private:
        ArrayList ^ someStrings;
    };
}

using namespace UsageLibrary;

void main()
{
    ArrayList^ newCollection = gcnew ArrayList(gcnew array<String^> {"a", "new", "collection"});

    WritableCollection^ collection = gcnew WritableCollection();

    // This line of code demonstrates how the entire collection
    // can be replaced by a property that's not read only.
    collection->SomeStrings = newCollection;

    // If the intent is to replace an entire collection,
    // implement and/or use the Clear() and AddRange() methods instead.
    collection->SomeStrings->Clear();
    collection->SomeStrings->AddRange(newCollection);
}
```

Related rules

- CA1819: Properties should not return arrays

CA2228: Do not ship unreleased resource formats

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2228
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A resource file was built using a version of .NET that is not currently supported.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Resource files that were built by using pre-release versions of .NET might not be usable by supported versions of .NET.

How to fix violations

To fix a violation of this rule, build the resource using a supported version of .NET.

When to suppress warnings

Do not suppress a warning from this rule.

CA2229: Implement serialization constructors

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2229
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

The type implements the System.Runtime.Serialization.ISerializable interface, is not a delegate or interface, and one of the following conditions is true:

- The type does not have a constructor that takes a SerializationInfo object and a StreamingContext object (the signature of the serialization constructor).
- The type is unsealed and the access modifier for its serialization constructor is not protected (family).
- The type is sealed and the access modifier for its serialization constructor is not private.

Rule description

This rule is relevant for types that support custom serialization. A type supports custom serialization if it implements the ISerializable interface. The serialization constructor is required to deserialize, or recreate, objects that have been serialized using the ISerializable.GetObjectData method.

How to fix violations

To fix a violation of this rule, implement the serialization constructor. For a sealed class, make the constructor private; otherwise, make it protected.

When to suppress warnings

Do not suppress a violation of the rule. The type will not be deserializable, and will not function in many scenarios.

Example

The following example shows a type that satisfies the rule.

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

namespace UsageLibrary
{
    [Serializable]
    public class SerializationConstructorsRequired : ISerializable
    {
        private int n1;

        // This is a regular constructor.
        public SerializationConstructorsRequired ()
        {
            n1 = -1;
        }
        // This is the serialization constructor.
        // Satisfies rule: ImplementSerializationConstructors.

        protected SerializationConstructorsRequired(
           SerializationInfo info,
           StreamingContext context)
        {
            n1 = (int) info.GetValue("n1", typeof(int));
        }

        // The following method serializes the instance.
        [SecurityPermission(SecurityAction.LinkDemand,
            Flags=SecurityPermissionFlag.SerializationFormatter)]
        void ISerializable.GetObjectData(SerializationInfo info,
           StreamingContext context)
        {
            info.AddValue("n1", n1);
        }
    }
}
```

Related rules

CA2237: Mark ISerializable types with SerializableAttribute

See also

- System.Runtime.Serialization.ISerializable
- System.Runtime.Serialization.SerializationInfo
- System.Runtime.Serialization.StreamingContext

CA2230: Use params for variable arguments

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2230
Category	Microsoft.Usage
Breaking change	Breaking

Cause

A public or protected type contains a public or protected method that uses the `VarArgs` calling convention.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

The `VarArgs` calling convention is used with certain method definitions that take a variable number of parameters. A method using the `VarArgs` calling convention is not Common Language Specification (CLS) compliant and might not be accessible across programming languages.

In C#, the `VarArgs` calling convention is used when a method's parameter list ends with the `__arglist` keyword. Visual Basic does not support the `VarArgs` calling convention, and C++ allows its use only in unmanaged code that uses the ellipse `...` notation.

How to fix violations

To fix a violation of this rule in C#, use the params keyword instead of `__arglist`.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows two methods, one that violates the rule and one that satisfies the rule.

```
using System;

[assembly: CLSCompliant(true)]
namespace UsageLibrary
{
    public class UseParams
    {
        // This method violates the rule.
        [CLSCompliant(false)]
        public void VariableArguments(__arglist)
        {
            ArgIterator argumentIterator = new ArgIterator(__arglist);
            for(int i = 0; i < argumentIterator.GetRemainingCount(); i++)
            {
                Console.WriteLine(
                    __refvalue(argumentIterator.GetNextArg(), string));
            }
        }

        // This method satisfies the rule.
        public void VariableArguments(params string[] wordList)
        {
            for(int i = 0; i < wordList.Length; i++)
            {
                Console.WriteLine(wordList[i]);
            }
        }
    }
}
```

See also

- System.Reflection.CallingConventions
- Language Independence and Language-Independent Components

CA2231: Overload operator equals on overriding ValueType.Equals

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2231
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A value type overrides System.Object.Equals but does not implement the equality operator.

By default, this rule only looks at externally visible types, but this is configurable.

Rule description

In most programming languages, there is no default implementation of the equality operator (==) for value types. If your programming language supports operator overloads, you should consider implementing the equality operator. Its behavior should be identical to that of Equals.

You cannot use the default equality operator in an overloaded implementation of the equality operator. Doing so will cause a stack overflow. To implement the equality operator, use the Object.Equals method in your implementation. For example:

```
If (Object.ReferenceEquals(left, Nothing)) Then
    Return Object.ReferenceEquals(right, Nothing)
Else
    Return left.Equals(right)
End If
```

```
if (Object.ReferenceEquals(left, null))
    return Object.ReferenceEquals(right, null);
return left.Equals(right);
```

How to fix violations

To fix a violation of this rule, implement the equality operator.

When to suppress warnings

It is safe to suppress a warning from this rule; however, we recommend that you provide the equality operator if possible.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca2231.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Usage). For more information, see Configure FxCop analyzers.

Example

The following example defines a type that violates this rule:

```
using System;

namespace UsageLibrary
{
    public struct PointWithoutHash
    {
        private int x,y;

        public PointWithoutHash(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

        public override string ToString()
        {
            return String.Format("({0},{1})",x,y);
        }

        public int X {get {return x;}}

        public int Y {get {return x;}}

        // Violates rule: OverrideGetHashCodeOnOverridingEquals.
        // Violates rule: OverrideOperatorEqualsOnOverridingValueTypeEquals.
        public override bool Equals (object obj)
        {
            if (obj.GetType() != typeof(PointWithoutHash))
                return false;

            PointWithoutHash p = (PointWithoutHash)obj;
            return ((this.x == p.x) && (this.y == p.y));
        }

    }

}
```

Related rules

- CA1046: Do not overload operator equals on reference types
- CA2225: Operator overloads have named alternates
- CA2226: Operators should have symmetrical overloads
- CA2224: Override equals on overloading operator equals
- CA2218: Override GetHashCode on overriding Equals

See also

- System.Object.Equals

CA2232: Mark Windows Forms entry points with STAThread

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2232
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An assembly references the System.Windows.Forms namespace, and its entry point is not marked with the System.STAThreadAttribute attribute.

Rule description

STAThreadAttribute indicates that the COM threading model for the application is single-threaded apartment. This attribute must be present on the entry point of any application that uses Windows Forms; if it is omitted, the Windows components might not work correctly. If the attribute is not present, the application uses the multithreaded apartment model, which is not supported for Windows Forms.

> **NOTE**
>
> Visual Basic projects that use the Application Framework do not have to mark the **Main** method with STAThread. The Visual Basic compiler does it automatically.

How to fix violations

To fix a violation of this rule, add the STAThreadAttribute attribute to the entry point. If the System.MTAThreadAttribute attribute is present, remove it.

When to suppress warnings

It is safe to suppress a warning from this rule if you are developing for the .NET Compact Framework, for which the STAThreadAttribute attribute is unnecessary and not supported.

Example

The following examples demonstrate the correct usage of STAThreadAttribute:

```
using System;
using  System.Windows.Forms;

namespace UsageLibrary
{
    public class MyForm: Form
    {
        public MyForm()
        {
            this.Text = "Hello World!";
        }

        // Satisfies rule: MarkWindowsFormsEntryPointsWithStaThread.
        [STAThread]
        public static void Main()
        {
            MyForm aform = new MyForm();
            Application.Run(aform);
        }
    }
}
```

```
Imports System
Imports System.Windows.Forms

NameSpace UsageLibrary

Public Class MyForm
   Inherits Form

   Public Sub New()
      Me.Text = "Hello World!"
   End Sub 'New

   ' Satisfies rule: MarkWindowsFormsEntryPointsWithStaThread.
   <STAThread()> _
   Public Shared Sub Main()
      Dim aform As New MyForm()
      Application.Run(aform)
   End Sub

End Class

End Namespace
```

CA2233: Operations should not overflow

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2233
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A method performs an arithmetic operation and does not validate the operands beforehand to prevent overflow.

> **NOTE**
> This legacy rule is not available in FxCop Analyzers. For more information, see Deprecated rules.

Rule description

Don't perform arithmetic operations without first validating the operands to make sure that the result of the operation is not outside the range of possible values for the data types involved. Depending on the execution context and the data types involved, arithmetic overflow can result in either a System.OverflowException or the most significant bits of the result discarded.

How to fix violations

To fix a violation of this rule, validate the operands before you perform the operation.

When to suppress warnings

It is safe to suppress a warning from this rule if the possible values of the operands will never cause the arithmetic operation to overflow.

Example of a violation

A method in the following example manipulates an integer that violates this rule. Visual Basic requires the **Remove** integer overflow option to be disabled for this to fire.

```
Imports System

Public Module Calculator

    Public Function Decrement(ByVal input As Integer) As Integer

        ' Violates this rule
        input = input - 1
        Return input

    End Function

End Module
```

```
using System;

namespace Samples
{
    public static class Calculator
    {
        public static int Decrement(int input)
        {
            // Violates this rule
            input--;
            return input;
        }
    }
}
```

If the method in this example is passed System.Int32.MinValue, the operation would underflow. This causes the most significant bit of the result to be discarded. The following code shows how this occurs.

```
public static void Main()
{
    int value = int.MinValue;    // int.MinValue is -2147483648
    value = Calculator.Decrement(value);
    Console.WriteLine(value);
}
```

```
Public Shared Sub Main()
    Dim value = Integer.MinValue    ' Integer.MinValue is -2147483648
    value = Calculator.Decrement(value)
    Console.WriteLine(value)
End Sub
```

Output:

```
2147483647
```

Fix with Input Parameter Validation

The following example fixes the previous violation by validating the value of input.

```
using System;

namespace Samples
{
    public static class Calculator
    {
        public static int Decrement(int input)
        {
            if (input == int.MinValue)
                throw new ArgumentOutOfRangeException(nameof(input), "input must be greater than
Int32.MinValue");

            input--;
            return input;
        }
    }
}
```

```
Public Module Calculator

    Public Function Decrement(ByVal input As Integer) As Integer

        If (input = Integer.MinValue) Then _
            Throw New ArgumentOutOfRangeException("input", "input must be greater than Int32.MinValue")

        input = input - 1
        Return input

    End Function

End Module
```

Fix with a Checked Block

The following example fixes the previous violation by wrapping the operation in a checked block. If the operation causes an overflow, a System.OverflowException will be thrown.

Checked blocks are not supported in Visual Basic.

```
using System;

namespace Samples
{
    public static class Calculator
    {
        public static int Decrement(int input)
        {
            checked
            {
                input--;
            }

            return input;
        }
    }
}
```

Turn on Checked Arithmetic Overflow/Underflow

If you turn on checked arithmetic overflow/underflow in C#, it is equivalent to wrapping every integer operation in

a checked block.

To turn on checked arithmetic overflow/underflow in C#:

1. In **Solution Explorer**, right-click your project and choose **Properties**.
2. Select the **Build** tab and click **Advanced**.
3. Select **Check for arithmetic overflow/underflow** and click **OK**.

See also

- System.OverflowException
- C# Operators
- Checked and Unchecked

CA2234: Pass System.Uri objects instead of strings

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2234
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A call is made to a method that has a string parameter whose name contains "uri", "Uri", "urn", "Urn", "url", or "Url" and the declaring type of the method contains a corresponding method overload that has a System.Uri parameter.

By default, this rule only looks at externally visible methods and types, but this is configurable.

Rule description

A parameter name is split into tokens based on the camel casing convention, and then each token is checked to see whether it equals "uri", "Uri", "urn", "Urn", "url", or "Url". If there is a match, the parameter is assumed to represent a uniform resource identifier (URI). A string representation of a URI is prone to parsing and encoding errors, and can lead to security vulnerabilities. The Uri class provides these services in a safe and secure manner. When there is a choice between two overloads that differ only regarding the representation of a URI, the user should choose the overload that takes a Uri argument.

How to fix violations

To fix a violation of this rule, call the overload that takes the Uri argument.

When to suppress warnings

It is safe to suppress a warning from this rule if the string parameter does not represent a URI.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), you can configure which parts of your codebase to run this rule on, based on their accessibility. For example, to specify that the rule should run only against the non-public API surface, add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.ca2234.api_surface = private, internal
```

You can configure this option for just this rule, for all rules, or for all rules in this category (Usage). For more information, see Configure FxCop analyzers.

Example

The following example shows a method, `ErrorProne`, that violates the rule and a method, `SaferWay`, that correctly calls the Uri overload:

```
Imports System

Namespace DesignLibrary

   Class History

      Friend Sub AddToHistory(uriString As String)
      End Sub

      Friend Sub AddToHistory(uriType As Uri)
      End Sub

   End Class

   Public Class Browser

      Dim uriHistory As New History()

      Sub ErrorProne()
         uriHistory.AddToHistory("http://www.adventure-works.com")
      End Sub

      Sub SaferWay()
         Try
            Dim newUri As New Uri("http://www.adventure-works.com")
            uriHistory.AddToHistory(newUri)
         Catch uriException As UriFormatException
         End Try
      End Sub

   End Class

End Namespace
```

```
#using <system.dll>
using namespace System;

namespace DesignLibrary
{
   ref class History
   {
   public:
      void AddToHistory(String^ uriString) {}
      void AddToHistory(Uri^ uriType) {}
   };

   public ref class Browser
   {
      History^ uriHistory;

   public:
      Browser()
      {
         uriHistory = gcnew History();
      }

      void ErrorProne()
      {
         uriHistory->AddToHistory("http://www.adventure-works.com");
      }

      void SaferWay()
      {
         try
         {
            Uri^ newUri = gcnew Uri("http://www.adventure-works.com");
            uriHistory->AddToHistory(newUri);
         }
         catch(UriFormatException^ uriException) {}
      }
   };
}
```

```
using System;

namespace DesignLibrary
{
   class History
   {
      internal void AddToHistory(string uriString) {}
      internal void AddToHistory(Uri uriType) {}
   }

   public class Browser
   {
      History uriHistory = new History();

      public void ErrorProne()
      {
         uriHistory.AddToHistory("http://www.adventure-works.com");
      }

      public void SaferWay()
      {
         try
         {
            Uri newUri = new Uri("http://www.adventure-works.com");
            uriHistory.AddToHistory(newUri);
         }
         catch(UriFormatException uriException) {}
      }
   }
}
```

Related rules

- CA1057: String URI overloads call System.Uri overloads
- CA1056: URI properties should not be strings
- CA1054: URI parameters should not be strings
- CA1055: URI return values should not be strings

CA2235: Mark all non-serializable fields

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2235
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An instance field of a type that is not serializable is declared in a type that is serializable.

Rule description

A serializable type is one that is marked with the System.SerializableAttribute attribute. When the type is serialized, a System.Runtime.Serialization.SerializationException exception is thrown if the type contains an instance field of a type that's not serializable *and* doesn't implement the System.Runtime.Serialization.ISerializable interface.

> **TIP**
> CA2235 does not fire for instance fields of types that implement ISerializable because they provide their own serialization logic.

How to fix violations

To fix a violation of this rule, apply the System.NonSerializedAttribute attribute to the field that is not serializable.

When to suppress warnings

Only suppress a warning from this rule if a System.Runtime.Serialization.ISerializationSurrogate type is declared that allows instances of the field to be serialized and deserialized.

Example

The following example shows two types: one that violates the rule and one that satisfies the rule.

```
using System;
using System.Runtime.Serialization;

namespace UsageLibrary
{
   public class Mouse
   {
      int buttons;
      string scanTypeValue;

      public int NumberOfButtons
      {
         get { return buttons; }
      }

      public string ScanType
      {
         get { return scanTypeValue; }
      }

      public Mouse(int numberOfButtons, string scanType)
      {
         buttons = numberOfButtons;
         scanTypeValue = scanType;
      }
   }

   [Serializable]
   public class InputDevices1
   {
      // Violates MarkAllNonSerializableFields.
      Mouse opticalMouse;

      public InputDevices1()
      {
         opticalMouse = new Mouse(5, "optical");
      }
   }

   [Serializable]
   public class InputDevices2
   {
      // Satisfies MarkAllNonSerializableFields.
      [NonSerialized]
      Mouse opticalMouse;

      public InputDevices2()
      {
         opticalMouse = new Mouse(5, "optical");
      }
   }
}
```

```
Imports System
Imports System.Runtime.Serialization

Namespace UsageLibrary

   Public Class Mouse

      Dim buttons As Integer
      Dim scanTypeValue As String

      ReadOnly Property NumberOfButtons As Integer
         Get
            Return buttons
         End Get
      End Property

      ReadOnly Property ScanType As String
         Get
            Return scanTypeValue
         End Get
      End Property

      Sub New(numberOfButtons As Integer, scanType As String)
         buttons = numberOfButtons
         scanTypeValue = scanType
      End Sub

   End Class

   <SerializableAttribute> _
   Public Class InputDevices1

      ' Violates MarkAllNonSerializableFields.
      Dim opticalMouse As Mouse

      Sub New()
         opticalMouse = New Mouse(5, "optical")
      End Sub

   End Class

   <SerializableAttribute> _
   Public Class InputDevices2

      ' Satisfies MarkAllNonSerializableFields.
      <NonSerializedAttribute> _
      Dim opticalMouse As Mouse

      Sub New()
         opticalMouse = New Mouse(5, "optical")
      End Sub

   End Class

End Namespace
```

Remarks

Rule CA2235 does not analyze types that implement the ISerializable interface (unless they are also marked with the SerializableAttribute attribute). This is because rule CA2237 already recommends marking types that implement the ISerializable interface with the SerializableAttribute attribute.

Related rules

- CA2229: Implement serialization constructors
- CA2236: Call base class methods on ISerializable types
- CA2237: Mark ISerializable types with SerializableAttribute
- CA2238: Implement serialization methods correctly
- CA2239: Provide deserialization methods for optional fields
- CA2240: Implement ISerializable correctly
- CA2120: Secure serialization constructors

CA2236: Call base class methods on ISerializable types

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2236
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A type derives from a type that implements the System.Runtime.Serialization.ISerializable interface, and one of the following conditions is true:

- The type implements the serialization constructor, that is, a constructor with the System.Runtime.Serialization.SerializationInfo, System.Runtime.Serialization.StreamingContext parameter signature, but does not call the serialization constructor of the base type.
- The type implements the System.Runtime.Serialization.ISerializable.GetObjectData method but does not call the GetObjectData method of the base type.

Rule description

In a custom serialization process, a type implements the GetObjectData method to serialize its fields and the serialization constructor to de-serialize the fields. If the type derives from a type that implements the ISerializable interface, the base type GetObjectData method and serialization constructor should be called to serialize/de-serialize the fields of the base type. Otherwise, the type will not be serialized and de-serialized correctly. Note that if the derived type does not add any new fields, the type does not need to implement the GetObjectData method nor the serialization constructor or call the base type equivalents.

How to fix violations

To fix a violation of this rule, call the base type GetObjectData method or serialization constructor from the corresponding derived type method or constructor.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows a derived type that satisfies the rule by calling the serialization constructor and GetObjectData method of the base class.

```
Imports System
Imports System.Runtime.Serialization
Imports System.Security.Permissions

Namespace UsageLibrary

   <SerializableAttribute> _
   Public Class BaseType
      Implements ISerializable

      Dim baseValue As Integer

      Sub New()
         baseValue = 3
      End Sub

      Protected Sub New( _
         info As SerializationInfo, context As StreamingContext)

         baseValue = info.GetInt32("baseValue")

      End Sub

      <SecurityPermissionAttribute(SecurityAction.Demand, _
          SerializationFormatter := True)> _
      Overridable Sub GetObjectData( _
         info As SerializationInfo, context As StreamingContext) _
         Implements ISerializable.GetObjectData

         info.AddValue("baseValue", baseValue)

      End Sub

   End Class

   <SerializableAttribute> _
   Public Class DerivedType: Inherits BaseType

      Dim derivedValue As Integer

      Sub New()
         derivedValue = 4
      End Sub

      Protected Sub New( _
         info As SerializationInfo, context As StreamingContext)

         MyBase.New(info, context)
         derivedValue = info.GetInt32("derivedValue")

      End Sub

      <SecurityPermissionAttribute(SecurityAction.Demand, _
          SerializationFormatter := True)> _
      Overrides Sub GetObjectData( _
         info As SerializationInfo, context As StreamingContext)

         info.AddValue("derivedValue", derivedValue)
         MyBase.GetObjectData(info, context)

      End Sub

   End Class

End Namespace
```

```
using System;
using System.Runtime.Serialization;
using System.Security.Permissions;

namespace UsageLibrary
{
   [SerializableAttribute]
   public class BaseType : ISerializable
   {
      int baseValue;

      public BaseType()
      {
         baseValue = 3;
      }

      protected BaseType(
         SerializationInfo info, StreamingContext context)
      {
         baseValue = info.GetInt32("baseValue");
      }

      [SecurityPermissionAttribute(SecurityAction.Demand,
          SerializationFormatter = true)]
      public virtual void GetObjectData(
         SerializationInfo info, StreamingContext context)
      {
         info.AddValue("baseValue", baseValue);
      }
   }

   [SerializableAttribute]
   public class DerivedType : BaseType
   {
      int derivedValue;

      public DerivedType()
      {
         derivedValue = 4;
      }

      protected DerivedType(
         SerializationInfo info, StreamingContext context) :
         base(info, context)
      {
         derivedValue = info.GetInt32("derivedValue");
      }

      [SecurityPermissionAttribute(SecurityAction.Demand,
          SerializationFormatter = true)]
      public override void GetObjectData(
         SerializationInfo info, StreamingContext context)
      {
         info.AddValue("derivedValue", derivedValue);
         base.GetObjectData(info, context);
      }
   }
}
```

Related rules

CA2240: Implement ISerializable correctly

CA2229: Implement serialization constructors

CA2238: Implement serialization methods correctly

CA2235: Mark all non-serializable fields

CA2237: Mark ISerializable types with SerializableAttribute

CA2239: Provide deserialization methods for optional fields

CA2120: Secure serialization constructors

CA2237: Mark ISerializable types with SerializableAttribute

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2237
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An externally visible type implements the System.Runtime.Serialization.ISerializable interface and the type is not marked with the System.SerializableAttribute attribute. The rule ignores derived types whose base type is not serializable.

Rule description

To be recognized by the common language runtime as serializable, types must be marked with the SerializableAttribute attribute even if the type uses a custom serialization routine through implementation of the ISerializable interface.

How to fix violations

To fix a violation of this rule, apply the SerializableAttribute attribute to the type.

When to suppress warnings

Do not suppress a warning from this rule for exception classes because they must be serializable to work correctly across application domains.

Example

The following example shows a type that violates the rule. Uncomment the SerializableAttribute attribute line to satisfy the rule.

```
Imports System
Imports System.Runtime.Serialization
Imports System.Security.Permissions

Namespace UsageLibrary

   ' <SerializableAttribute> _
   Public Class BaseType
      Implements ISerializable

      Dim baseValue As Integer

      Sub New()
         baseValue = 3
      End Sub

      Protected Sub New( _
         info As SerializationInfo, context As StreamingContext)

         baseValue = info.GetInt32("baseValue")

      End Sub

      <SecurityPermissionAttribute(SecurityAction.Demand, _
          SerializationFormatter := True)> _
      Overridable Sub GetObjectData( _
         info As SerializationInfo, context As StreamingContext) _
         Implements ISerializable.GetObjectData

         info.AddValue("baseValue", baseValue)

      End Sub

   End Class

End Namespace
```

```
using System;
using System.Runtime.Serialization;
using System.Security.Permissions;

namespace UsageLibrary
{
   // [SerializableAttribute]
   public class BaseType : ISerializable
   {
      int baseValue;

      public BaseType()
      {
         baseValue = 3;
      }

      protected BaseType(
         SerializationInfo info, StreamingContext context)
      {
         baseValue = info.GetInt32("baseValue");
      }

      [SecurityPermissionAttribute(SecurityAction.Demand,
          SerializationFormatter = true)]
      public virtual void GetObjectData(
         SerializationInfo info, StreamingContext context)
      {
         info.AddValue("baseValue", baseValue);
      }
   }
}
```

Related rules

CA2236: Call base class methods on ISerializable types

CA2240: Implement ISerializable correctly

CA2229: Implement serialization constructors

CA2238: Implement serialization methods correctly

CA2235: Mark all non-serializable fields

CA2239: Provide deserialization methods for optional fields

CA2120: Secure serialization constructors

CA2238: Implement serialization methods correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2238
Category	Microsoft.Usage
Breaking change	Breaking - If the method is visible outside the assembly. Non-breaking - If the method is not visible outside the assembly.

Cause

A method that handles a serialization event does not have the correct signature, return type, or visibility.

Rule description

A method is designated a serialization event handler by applying one of the following serialization event attributes:

- System.Runtime.Serialization.OnSerializingAttribute
- System.Runtime.Serialization.OnSerializedAttribute
- System.Runtime.Serialization.OnDeserializingAttribute
- System.Runtime.Serialization.OnDeserializedAttribute

  Serialization event handlers take a single parameter of type System.Runtime.Serialization.StreamingContext, return `void`, and have `private` visibility.

How to fix violations

To fix a violation of this rule, correct the signature, return type, or visibility of the serialization event handler.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows correctly declared serialization event handlers.

```
Imports System
Imports System.Runtime.Serialization

Namespace UsageLibrary

   <SerializableAttribute> _
   Public Class SerializationEventHandlers

      <OnSerializingAttribute> _
      Private Sub OnSerializing(context As StreamingContext)
      End Sub

      <OnSerializedAttribute> _
      Private Sub OnSerialized(context As StreamingContext)
      End Sub

      <OnDeserializingAttribute> _
      Private Sub OnDeserializing(context As StreamingContext)
      End Sub

      <OnDeserializedAttribute> _
      Private Sub OnDeserialized(context As StreamingContext)
      End Sub

   End Class

End Namespace
```

```
using System;
using System.Runtime.Serialization;

namespace UsageLibrary
{
   [SerializableAttribute]
   public class SerializationEventHandlers
   {
      [OnSerializingAttribute]
      void OnSerializing(StreamingContext context) {}

      [OnSerializedAttribute]
      void OnSerialized(StreamingContext context) {}

      [OnDeserializingAttribute]
      void OnDeserializing(StreamingContext context) {}

      [OnDeserializedAttribute]
      void OnDeserialized(StreamingContext context) {}
   }
}
```

Related rules

CA2236: Call base class methods on ISerializable types

CA2240: Implement ISerializable correctly

CA2229: Implement serialization constructors

CA2235: Mark all non-serializable fields

CA2237: Mark ISerializable types with SerializableAttribute

CA2239: Provide deserialization methods for optional fields

CA2120: Secure serialization constructors

CA2239: Provide deserialization methods for optional fields

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2239
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A type has a field that is marked with the System.Runtime.Serialization.OptionalFieldAttribute attribute and the type does not provide de-serialization event handling methods.

Rule description

The OptionalFieldAttribute attribute has no effect on serialization; a field marked with the attribute is serialized. However, the field is ignored on de-serialization and retains the default value associated with its type. De-serialization event handlers should be declared to set the field during the de-serialization process.

How to fix violations

To fix a violation of this rule, add de-serialization event handling methods to the type.

When to suppress warnings

It is safe to suppress a warning from this rule if the field should be ignored during the de-serialization process.

Example

The following example shows a type with an optional field and de-serialization event handling methods.

```
using System;
using System.Reflection;
using System.Runtime.Serialization;

[assembly: AssemblyVersionAttribute("2.0.0.0")]
namespace UsageLibrary
{
   [SerializableAttribute]
   public class SerializationEventHandlers
   {
      [OptionalFieldAttribute(VersionAdded = 2)]
      int optionalField = 5;

      [OnDeserializingAttribute]
      void OnDeserializing(StreamingContext context)
      {
         optionalField = 5;
      }

      [OnDeserializedAttribute]
      void OnDeserialized(StreamingContext context)
      {
         // Set optionalField if dependent on other deserialized values.
      }
   }
}
```

```
Imports System
Imports System.Reflection
Imports System.Runtime.Serialization

<Assembly: AssemblyVersionAttribute("2.0.0.0")>
Namespace UsageLibrary

   <SerializableAttribute> _
   Public Class SerializationEventHandlers

      <OptionalFieldAttribute(VersionAdded := 2)> _
      Dim optionalField As Integer = 5

      <OnDeserializingAttribute> _
      Private Sub OnDeserializing(context As StreamingContext)
         optionalField = 5
      End Sub

      <OnDeserializedAttribute> _
      Private Sub OnDeserialized(context As StreamingContext)
         ' Set optionalField if dependent on other deserialized values.
      End Sub

   End Class

End Namespace
```

Related rules

CA2236: Call base class methods on ISerializable types

CA2240: Implement ISerializable correctly

CA2229: Implement serialization constructors

CA2238: Implement serialization methods correctly

CA2235: Mark all non-serializable fields

CA2237: Mark ISerializable types with SerializableAttribute

CA2120: Secure serialization constructors

CA2240: Implement ISerializable correctly

8/4/2020 • 6 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2240
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An externally visible type is assignable to the System.Runtime.Serialization.ISerializable interface and one of the following conditions is true:

- The type inherits but does not override the System.Runtime.Serialization.ISerializable.GetObjectData method and the type declares instance fields that are not marked with the System.NonSerializedAttribute attribute.
- The type is not sealed and the type implements a GetObjectData method that is not externally visible and overridable.

Rule description

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.

How to fix violations

To fix a violation of this rule, make the GetObjectData method visible and overridable and make sure all instance fields are included in the serialization process or explicitly marked with the NonSerializedAttribute attribute.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows two serializable types that violate the rule.

```
using System;
using System.Security.Permissions;
using System.Runtime.Serialization;

namespace Samples1
{
    // Violates this rule
    [Serializable]
    public class Book : ISerializable
    {
        private readonly string _Text;

        public Book(string text)
        {
            if (text == null)
                throw new ArgumentNullException("text");

            _Text = text;
        }

        protected Book(SerializationInfo info, StreamingContext context)
        {
            if (info == null)
                throw new ArgumentNullException("info");

            _Text = info.GetString("Text");
        }

        public string Text
        {
            get { return _Text; }
        }

        [SecurityPermission(SecurityAction.Demand, SerializationFormatter = true)]
        public void GetObjectData(SerializationInfo info, StreamingContext context)
        {
            if (info == null)
                throw new ArgumentNullException("info");

            info.AddValue("Text", _Text);
        }
    }

    // Violates this rule
    [Serializable]
    public class LibraryBook : Book
    {
        private readonly DateTime _CheckedOut;

        public LibraryBook(string text, DateTime checkedOut)
            : base(text)
        {
            _CheckedOut = checkedOut;
        }

        public DateTime CheckedOut
        {
            get { return _CheckedOut; }
        }
    }
}
```

```
using namespace System;
using namespace System::Security::Permissions;
using namespace System::Runtime::Serialization;
```

```
namespace Samples1
{
    // Violates this rule
    [Serializable]
    public ref class Book : ISerializable
    {
    private:
        initonly String^ _Title;

    public:
        Book(String^ title)
        {
            if (title == nullptr)
                throw gcnew ArgumentNullException("title");

            _Title = title;
        }

        property String^ Title
        {
            String^ get()
            {
                return _Title;
            }
        }

    protected:
        Book(SerializationInfo^ info, StreamingContext context)
        {
            if (info == nullptr)
                throw gcnew ArgumentNullException("info");

            _Title = info->GetString("Title");
        }

    private:
        [SecurityPermission(SecurityAction::LinkDemand, Flags =
SecurityPermissionFlag::SerializationFormatter)]
        void virtual GetObjectData(SerializationInfo^ info, StreamingContext context) sealed =
ISerializable::GetObjectData
        {
            if (info == nullptr)
                throw gcnew ArgumentNullException("info");

            info->AddValue("Title", _Title);
        }
    };

    // Violates this rule
    [Serializable]
    public ref class LibraryBook : Book
    {
        initonly DateTime _CheckedOut;

    public:
        LibraryBook(String^ title, DateTime checkedOut) : Book(title)
        {
            _CheckedOut = checkedOut;
        }

        property DateTime CheckedOut
        {
            DateTime get()
            {
                return _CheckedOut;
            }
        }
    };
}
```

```
Imports System
Imports System.Security.Permissions
Imports System.Runtime.Serialization

Namespace Samples1

    ' Violates this rule
    <Serializable()> _
    Public Class Book
        Implements ISerializable

        Private ReadOnly _Title As String

        Public Sub New(ByVal title As String)
            If (title Is Nothing) Then Throw New ArgumentNullException("title")
            _Title = title
        End Sub

        Protected Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
            If (info Is Nothing) Then Throw New ArgumentNullException("info")

            _Title = info.GetString("Title")
        End Sub

        Public ReadOnly Property Title() As String
            Get
                Return _Title
            End Get
        End Property

        <SecurityPermissionAttribute(SecurityAction.LinkDemand,
Flags:=SecurityPermissionFlag.SerializationFormatter)> _
        Public Sub GetObjectData(ByVal info As SerializationInfo, ByVal context As StreamingContext) _
            Implements ISerializable.GetObjectData

            If (info Is Nothing) Then Throw New ArgumentNullException("info")

            info.AddValue("Title", _Title)
        End Sub
    End Class

    ' Violates this rule
    <Serializable()> _
    Public Class LibraryBook
        Inherits Book

        Private ReadOnly _CheckedOut As Date

        Public Sub New(ByVal text As String, ByVal checkedOut As Date)
            MyBase.New(text)
            _CheckedOut = checkedOut
        End Sub

        Public ReadOnly Property CheckedOut() As Date
            Get
                Return _CheckedOut
            End Get
        End Property

    End Class
End Namespace
```

Example

The following example fixes the two previous violations by providing an overrideable implementation of GetObjectData on the Book class and by providing an implementation of `GetObjectData` on the Library class.

```
using namespace System;
using namespace System::Security::Permissions;
using namespace System::Runtime::Serialization;

namespace Samples2
{
    [Serializable]
    public ref class Book : ISerializable
    {
    private:
        initonly String^ _Title;

    public:
        Book(String^ title)
        {
            if (title == nullptr)
                throw gcnew ArgumentNullException("title");

            _Title = title;
        }

        property String^ Title
        {
            String^ get()
            {
                return _Title;
            }
        }

    protected:
        Book(SerializationInfo^ info, StreamingContext context)
        {
            if (info == nullptr)
                throw gcnew ArgumentNullException("info");

            _Title = info->GetString("Title");
        }

        [SecurityPermission(SecurityAction::LinkDemand, Flags =
SecurityPermissionFlag::SerializationFormatter)]
        void virtual GetObjectData(SerializationInfo^ info, StreamingContext context) =
ISerializable::GetObjectData
        {
            if (info == nullptr)
                throw gcnew ArgumentNullException("info");

            info->AddValue("Title", _Title);
        }
    };

    [Serializable]
    public ref class LibraryBook : Book
    {
        initonly DateTime _CheckedOut;

    public:
        LibraryBook(String^ title, DateTime checkedOut)
            : Book(title)
        {
            _CheckedOut = checkedOut;
        }
```

```
        property DateTime CheckedOut
        {
            DateTime get()
            {
                return _CheckedOut;
            }
        }

    protected:
        LibraryBook(SerializationInfo^ info, StreamingContext context) : Book(info, context)
        {
            _CheckedOut = info->GetDateTime("CheckedOut");
        }

        [SecurityPermission(SecurityAction::LinkDemand, Flags =
SecurityPermissionFlag::SerializationFormatter)]
        void virtual GetObjectData(SerializationInfo^ info, StreamingContext context) override
        {
            Book::GetObjectData(info, context);
            info->AddValue("CheckedOut", _CheckedOut);
        }
    };
}
```

```
using System;
using System.Security.Permissions;
using System.Runtime.Serialization;

namespace Samples2
{
    [Serializable]
    public class Book : ISerializable
    {
        private readonly string _Title;

        public Book(string title)
        {
            if (title == null)
                throw new ArgumentNullException("title");

            _Title = title;
        }

        protected Book(SerializationInfo info, StreamingContext context)
        {
            if (info == null)
                throw new ArgumentNullException("info");

            _Title = info.GetString("Title");
        }

        public string Title
        {
            get { return _Title; }
        }

        [SecurityPermission(SecurityAction.Demand, SerializationFormatter = true)]
        protected virtual void GetObjectData(SerializationInfo info, StreamingContext context)
        {
            info.AddValue("Title", _Title);
        }

        [SecurityPermission(SecurityAction.LinkDemand, Flags =
SecurityPermissionFlag.SerializationFormatter)]
        void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
        {
            if (info == null)
```

```
                throw new ArgumentNullException("info");

            GetObjectData(info, context);
        }
    }

    [Serializable]
    public class LibraryBook : Book
    {
        private readonly DateTime _CheckedOut;

        public LibraryBook(string title, DateTime checkedOut)
            : base(title)
        {
            _CheckedOut = checkedOut;
        }

        protected LibraryBook(SerializationInfo info, StreamingContext context)
            : base(info, context)
        {
            _CheckedOut = info.GetDateTime("CheckedOut");
        }

        public DateTime CheckedOut
        {
            get { return _CheckedOut; }
        }

        [SecurityPermission(SecurityAction.Demand, SerializationFormatter = true)]
        protected override void GetObjectData(SerializationInfo info, StreamingContext context)
        {
            base.GetObjectData(info, context);

            info.AddValue("CheckedOut", _CheckedOut);
        }
    }
}
```

```
Imports System
Imports System.Security.Permissions
Imports System.Runtime.Serialization

Namespace Samples2

    <Serializable()> _
    Public Class Book
        Implements ISerializable

        Private ReadOnly _Title As String

        Public Sub New(ByVal title As String)
            If (title Is Nothing) Then Throw New ArgumentNullException("title")
            _Title = title
        End Sub

        Protected Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
            If (info Is Nothing) Then Throw New ArgumentNullException("info")

            _Title = info.GetString("Title")
        End Sub

        Public ReadOnly Property Title() As String
            Get
                Return _Title
            End Get
        End Property
```

```
        <SecurityPermissionAttribute(SecurityAction.LinkDemand,
Flags:=SecurityPermissionFlag.SerializationFormatter)> _
        Protected Overridable Sub GetObjectData(ByVal info As SerializationInfo, ByVal context As
StreamingContext) _
            Implements ISerializable.GetObjectData

            If (info Is Nothing) Then Throw New ArgumentNullException("info")

            info.AddValue("Title", _Title)
        End Sub
    End Class


    <Serializable()> _
    Public Class LibraryBook
        Inherits Book

        Private ReadOnly _CheckedOut As Date

        Public Sub New(ByVal text As String, ByVal checkedOut As Date)
            MyBase.New(text)
            _CheckedOut = checkedOut
        End Sub

        Protected Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
            MyBase.New(info, context)

            _CheckedOut = info.GetDateTime("CheckedOut")
        End Sub

        Public ReadOnly Property CheckedOut() As Date
            Get
                Return _CheckedOut
            End Get
        End Property

        <SecurityPermissionAttribute(SecurityAction.LinkDemand,
Flags:=SecurityPermissionFlag.SerializationFormatter)> _
        Protected Overrides Sub GetObjectData(ByVal info As System.Runtime.Serialization.SerializationInfo,
_
                                              ByVal context As
System.Runtime.Serialization.StreamingContext)

            MyBase.GetObjectData(info, context)

            info.AddValue("CheckedOut", _CheckedOut)
        End Sub
    End Class
End Namespace
```

Related rules

- CA2236: Call base class methods on ISerializable types
- CA2229: Implement serialization constructors
- CA2238: Implement serialization methods correctly
- CA2235: Mark all non-serializable fields
- CA2237: Mark ISerializable types with SerializableAttribute
- CA2239: Provide deserialization methods for optional fields
- CA2120: Secure serialization constructors

CA2241: Provide correct arguments to formatting methods

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2241
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

The `format` string argument passed to a method such as WriteLine, Write, or System.String.Format does not contain a format item that corresponds to each object argument, or vice versa.

By default, this rule only analyzes calls to the three methods mentioned above, but this is configurable.

Rule description

The arguments to methods such as WriteLine, Write, and Format consist of a format string followed by several System.Object instances. The format string consists of text and embedded format items of the form, {index[,alignment][:formatString]}. 'index' is a zero-based integer that indicates which of the objects to format. If an object does not have a corresponding index in the format string, the object is ignored. If the object specified by 'index' does not exist, a System.FormatException is thrown at run time.

How to fix violations

To fix a violation of this rule, provide a format item for each object argument and provide an object argument for each format item.

When to suppress warnings

Do not suppress a warning from this rule.

Configurability

If you're running this rule from FxCop analyzers (and not with legacy analysis), this rule is configurable.

Additional string formatting methods

You can configure names of additional string formatting methods which should be analyzed by this rule. For example, to specify all methods named `MyFormat` as string formatting methods, you can add the following key-value pair to an .editorconfig file in your project:

```
dotnet_code_quality.CA2241.additional_string_formatting_methods = MyFormat
```

Allowed method name formats in the option value (separated by '|'):

- Method name only (includes all methods with the name, regardless of the containing type or namespace)
- Fully qualified names in the symbol's documentation ID format, with an optional `M:` prefix.

Examples:

OPTION VALUE	SUMMARY
`dotnet_code_quality.CA2241.additional_string_formatting_methods = MyFormat`	Matches all methods named 'MyFormat' in the compilation
`dotnet_code_quality.CA2241.additional_string_formatting_methods = MyFormat1|MyFormat2`	Matches all methods named either 'MyFormat1' or 'MyFormat2' in the compilation
`dotnet_code_quality.CA2241.additional_string_formatting_methods = NS.MyType.MyFormat(ParamType)`	Matches specific method 'MyFormat' with given fully qualified signature
`dotnet_code_quality.CA2241.additional_string_formatting_methods = NS1.MyType1.MyFormat1(ParamType)|NS2.MyType2.MyFormat2(ParamType)`	Matches specific methods 'MyFormat1' and 'MyFormat2' with respective fully qualified signature

You can configure this option for just this rule, for all rules, or for all rules in this category (Usage). For more information, see Configure FxCop analyzers.

Alternatively, instead of specifying an explicit list of additional string formatting methods, you can configure the analyzer to automatically attempt to determine the string formatting method. By default, this option is disabled. If the option is enabled, any method that has a `string format` parameter followed by a `params object[]` parameter is considered a string formatting method:

```
dotnet_code_quality.CA2241.try_determine_additional_string_formatting_methods_automatically = true
```

Example

The following example shows two violations of the rule.

```
Imports System

Namespace UsageLibrary

   Class CallsStringFormat

      Sub CallFormat()

         Dim file As String = "file name"
         Dim errors As Integer = 13

         ' Violates the rule.
         Console.WriteLine(String.Format("{0}", file, errors))

         Console.WriteLine(String.Format("{0}: {1}", file, errors))

         ' Violates the rule and generates a FormatException at runtime.
         Console.WriteLine(String.Format("{0}: {1}, {2}", file, errors))

      End Sub

   End Class

End Namespace
```

```
using System;

namespace UsageLibrary
{
   class CallsStringFormat
   {
      void CallFormat()
      {
         string file = "file name";
         int errors = 13;

         // Violates the rule.
         Console.WriteLine(string.Format("{0}", file, errors));

         Console.WriteLine(string.Format("{0}: {1}", file, errors));

         // Violates the rule and generates a FormatException at runtime.
         Console.WriteLine(string.Format("{0}: {1}, {2}", file, errors));
      }
   }
}
```

CA2242: Test for NaN correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2242
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An expression tests a value against System.Single.NaN or System.Double.NaN.

Rule description

System.Double.NaN, which represents not-a-number, results when an arithmetic operation is undefined. Any expression that tests equality between a value and System.Double.NaN always returns `false`. Any expression that tests inequality between a value and System.Double.NaN always returns `true`.

How to fix violations

To fix a violation of this rule and accurately determine whether a value represents System.Double.NaN, use System.Single.IsNaN or System.Double.IsNaN to test the value.

When to suppress warnings

Do not suppress a warning from this rule.

Example

The following example shows two expressions that incorrectly test a value against System.Double.NaN and an expression that correctly uses System.Double.IsNaN to test the value.

```
Imports System

Namespace UsageLibrary

   Class NaNTests

      Shared zero As Double

      Shared Sub Main()
         Console.WriteLine( 0/zero = Double.NaN )
         Console.WriteLine( 0/zero <> Double.NaN )
         Console.WriteLine( Double.IsNaN(0/zero) )
      End Sub

   End Class

End Namespace
```

```
using System;

namespace UsageLibrary
{
   class NaNTests
   {
      static double zero;

      static void Main()
      {
         Console.WriteLine( 0/zero == double.NaN );
         Console.WriteLine( 0/zero != double.NaN );
         Console.WriteLine( double.IsNaN(0/zero) );
      }
   }
}
```

CA2243: Attribute string literals should parse correctly

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2243
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An attribute's string literal parameter does not parse correctly for a URL, GUID, or Version.

Rule description

Since attributes are derived from System.Attribute, and attributes are used at compile time, only constant values can be passed to their constructors. Attribute parameters that must represent URLs, GUIDs, and Versions cannot be typed as System.Uri, System.Guid, and System.Version, because these types cannot be represented as constants. Instead, they must be represented by strings.

Because the parameter is typed as a string, it is possible that an incorrectly formatted parameter could be passed at compile time.

This rule uses a naming heuristic to find parameters that represent a uniform resource identifier (URI), a Globally Unique Identifier (GUID), or a Version, and verifies that the passed value is correct.

How to fix violations

Change the parameter string to a correctly formed URL, GUID, or Version.

When to suppress warnings

It is safe to suppress a warning from this rule if the parameter does not represent a URL, GUID, or Version.

Example

The following example shows code for the AssemblyFileVersionAttribute that violates this rule.

```
using System;
using System.Runtime.InteropServices;

namespace Samples
{
    [AttributeUsage(AttributeTargets.Assembly, Inherited = false)]
    [ComVisible(true)]
    public sealed class AssemblyFileVersionAttribute : Attribute
    {
        public AssemblyFileVersionAttribute(string version) { }

        public string Version { get; set; }
    }

    // Since the parameter is typed as a string, it is possible
    // to pass an invalid version number at compile time. The rule
    // would be violated by the following code: [assembly : AssemblyFileVersion("xxxxx")]
}
```

The rule is triggered by the following parameters:

- Parameters that contain 'version' and cannot be parsed to System.Version.
- Parameters that contain 'guid' and cannot be parsed to System.Guid.
- Parameters that contain 'uri', 'urn', or 'url' and cannot be parsed to System.Uri.

See also

- CA1054: URI parameters should not be strings

CA2244: Do not duplicate indexed element initializations

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2244
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

An object initializer has more than one indexed element initializer with the same constant index. All but the last initializer are redundant.

Rule description

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements.

Indexed element initializers in objects initializers must initialize unique elements. A duplicate index will overwrite a previous element initialization.

> **NOTE**
> Rule CA2244 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.8.

How to fix violations

To fix violations, remove all the redundant indexed element initializers that are overwritten by any of the subsequent element initializer(s). For example, the following code snippet shows a violation of the rule and couple of potential fixes:

```
using System.Collections.Generic;

class C
{
    public void M()
    {
        var dictionary = new Dictionary<int, int>
        {
            [1] = 1, // CA2244
            [2] = 2,
            [1] = 3
        };
    }
}
```

```
using System.Collections.Generic;

class C
{
    public void M()
    {
        var dictionary = new Dictionary<int, int>
        {
            [2] = 2,
            [1] = 3
        };
    }
}
```

```
using System.Collections.Generic;

class C
{
    public void M()
    {
        var dictionary = new Dictionary<int, int>
        {
            [1] = 1,
            [2] = 2
        };
    }
}
```

TIP

A code fix is available for this rule in Visual Studio. To use it, position the cursor on the violation and press **Ctrl+.** (period). Choose **Remove redundant element initializer** from the list of options that's presented.

When to suppress warnings

Do not suppress violations for this rule.

See also

- Usage warnings

CA2245: Do not assign a property to itself

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2245
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A property was accidentally assigned to itself.

Rule description

C# compiler generates a warning CS1717: Assignment made to same variable; did you mean to assign something else? when a field, local or parameter symbol is assigned to itself. Such a mistake is common when a local, parameter, or field symbol has a name similar to another symbol in scope. Instead of using different symbols on the left-hand and right-hand side of the assignment, the same symbol was used on both sides. This leads to a redundant assignment of the value to itself and generally indicates a functional bug.

Assigning a property to itself is also a similar functional bug for almost all real world cases. However, in some extreme corner cases, fetching a property value can have side effects and the property's new value is different from the original value. If so, property self-assignment is not redundant and cannot be removed. This prevents the compiler from generating a `CS1717` warning for property self-assignment, without introducing a breaking change for these cases.

Rule `CA2245` aims at filling this gap. It reports the violation for property self-assignment to help fix these functional bugs. For the small set of corner cases where property self-assignment is desirable, `CA2245` violations can be suppressed in source with an appropriate justification comment.

> **NOTE**
>
> Rule CA2245 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.8.

How to fix violations

To fix violations, use different symbols on the left-hand and the right-hand side of the assignment. For example, the following code snippet shows a violation of the rule and how to fix it:

```
public class C
{
    private int p = 0;
    public int P { get; private set; }

    public void M(int p)
    {
        // CS1717: Accidentally assigned the parameter 'p' to itself.
        p = p;

        // CA2245: Accidentally assigned the property 'P' to itself.
        P = P;
    }
}
```

```
public class C
{
    private int p = 0;
    public int P { get; private set; }

    public void M(int p)
    {
        // No violation, now the parameter is assigned to the field.
        this.p = p;

        // No violation, now the parameter is assigned to the property.
        P = p;
    }
}
```

When to suppress warnings

It is fine to suppress violations from this rule if fetching a property value can have side effects and the property's new value is different from the original value. If so, property self-assignment is not redundant. A justification comment should be added to the suppression to document this as expected behavior.

Related rules

- CS1717: Assignment made to same variable; did you mean to assign something else?
- CA2011: Do not assign property within its setter
- CA2246: Do not assign a symbol and its member in the same statement

See also

- Usage warnings

CA2246: Do not assign a symbol and its member in the same statement

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2246
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

A symbol and its member were assigned in the same statement. For example:

```
// 'a' and 'a.Field' are assigned in the same statement
a.Field = a = b;
```

Rule description

Assigning a symbol and its member, that is, a field or a property, in the same statement is not recommended. It is not clear if the member access was intended to use the symbol's old value prior to the assignment or the new value from the assignment in this statement. For clarity, the multi-assign statement must be split into two or more simple assignment statements.

> **NOTE**
>
> Rule CA2246 is not available in legacy analysis. It was first introduced in FxCop analyzers version 2.9.6.

How to fix violations

To fix violations, split the multi-assign statement into two or more simple assignment statements. For example, the following code snippet shows a violation of the rule and a couple of ways to fix it based on the user intent:

```
public class C
{
    public C Field;
}

public class Test
{
    public void M(C a, C b)
    {
        // Let us assume 'a' points to 'Instance1' and 'b' points to 'Instance2' at the start of the method.
        // It is not clear if the user intent in the below statement is to assign to 'Instance1.Field' or
'Instance2.Field'.
        // CA2246: Symbol 'a' and its member 'Field' are both assigned in the same statement. You are at risk
of assigning the member of an unintended object.
        a.Field = a = b;
    }
}
```

```
public class C
{
    public C Field;
}

public class Test
{
    public void M(C a, C b)
    {
        // Let us assume 'a' points to 'Instance1' and 'b' points to 'Instance2' at the start of the method.
        // 'Instance1.Field' is intended to be assigned.
        var instance1 = a;
        a = b;
        instance1.Field = a;
    }
}
```

```
public class C
{
    public C Field;
}

public class Test
{
    public void M(C a, C b)
    {
        // Let us assume 'a' points to 'Instance1' and 'b' points to 'Instance2' at the start of the method.
        // 'Instance2.Field' is intended to be assigned.
        a = b;
        b.Field = a; // or 'a.Field = a;'
    }
}
```

When to suppress warnings

Do not suppress violations from this rule.

Related rules

- CA2245: Do not assign a property to itself

See also

- Usage warnings

CA2247: Argument passed to TaskCompletionSource constructor should be TaskCreationOptions enum instead of TaskContinuationOptions enum

8/4/2020 • 2 minutes to read • Edit Online

CheckId	CA2247
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

Constructing a `System.Threading.Tasks.TaskCompletionSource` with a System.Threading.Tasks.TaskContinuationOptions enum value rather than a System.Threading.Tasks.TaskCreationOptions enum value. Using System.Object.ReferenceEquals method to test one or more value types for equality.

Rule description

The TaskCompletionSource type has a constructor that accepts a System.Threading.Tasks.TaskCreationOptions enum value, and another constructor that accepts a Object. Accidentally passing a System.Threading.Tasks.TaskContinuationOptions enum value instead of a System.Threading.Tasks.TaskCreationOptions enum value will result in calling the Object-based constructor: it will compile and run, but it will not have the intended behavior.

How to fix violations

To fix the violation, replace the System.Threading.Tasks.TaskContinuationOptions enum value with the corresponding System.Threading.Tasks.TaskCreationOptions enum value.

```
// Violation
var tcs = new TaskCompletionSource<int>(TaskContinuationOptions.RunContinuationsAsynchronously);

// Fixed
var tcs = new TaskCompletionSource<int>(TaskCreationOptions.RunContinuationsAsynchronously);
```

When to suppress warnings

A violation of this rule almost always highlights a bug in the calling code, such that the code will not behave as the developer intended, with the TaskCompletionSource effectively ignoring the specified option. The only time it is safe to suppress the warning is if the developer actually intended to pass a boxed System.Threading.Tasks.TaskContinuationOptions as the object state argument to the TaskCompletionSource.

See also

- Usage warnings

CA2248: Provide correct enum argument to Enum.HasFlag

8/4/2020 • 2 minutes to read • Edit Online

ITEM	VALUE
CheckId	CA2248
Category	Microsoft.Usage
Breaking change	Non-breaking

Cause

The enum type passed as an argument to the `HasFlag` method call is different from the calling enum type.

Rule description

The `Enum.HasFlag` method expects the `enum` argument to be of the same `enum` type as the instance on which the method is invoked. If these are different `enum` types, an unhandled exception will be thrown at runtime.

> **NOTE**
> Rule CA2248 is not available in legacy analysis. It was first introduced in FxCop analyzers version 3.1.0.

How to fix violations

To fix violations, use the same enum type on both the argument and the caller:

```
public class C
{
    [Flags]
    public enum MyEnum { A, B, }

    [Flags]
    public enum OtherEnum { A, }

    public void Method(MyEnum m)
    {
        m.HasFlag(OtherEnum.A); // Enum types are different, this call will cause an `ArgumentException` to be
thrown at runtime

        m.HasFlag(MyEnum.A); // Valid call
    }
}
```

When to suppress warnings

Do not suppress violations from this rule.

Automatic feature suspension

8/4/2020 • 2 minutes to read • Edit Online

If your available system memory falls to 200 MB or less, Visual Studio displays the following message in the code editor:

When Visual Studio detects a low memory condition, it automatically suspends certain advanced features to help it remain stable. Visual Studio continues to work as before, but its performance is degraded.

In a low memory condition, the following actions take place:

- Live code analysis for Visual C# and Visual Basic is reduced to minimal scope.
- Garbage Collection (GC) low-latency mode for Visual C# and Visual Basic is disabled.
- Visual Studio caches are flushed.

Improve Visual Studio performance

For tips and tricks on how to improve Visual Studio performance when dealing with large solutions or low-memory conditions, see Performance considerations for large solutions.

Live code analysis is reduced to minimal scope

By default, live code analysis executes for open documents and projects. You can customize this analysis scope to be reduced to current document or increased to entire solution. For more information, see How to: Configure live code analysis scope for managed code. In a low memory condition, Visual Studio forces the live analysis scope to be reduced to current document. However, you can re-enable your preferred analysis scope by choosing the **Re-enable** button in the info bar when it appears or by restarting Visual Studio. The Options dialog box always shows the current live code analysis scope settings.

GC low-latency disabled

To re-enable GC low-latency mode, restart Visual Studio. By default, Visual Studio enables GC low-latency mode whenever you are typing to ensure that your typing doesn't block any GC operations. However, if a low memory condition causes Visual Studio to display the automatic suspension warning, GC low-latency mode is disabled for that session. Restarting Visual Studio re-enables the default GC behavior. For more information, see GCLatencyMode.

Visual Studio caches flushed

If you continue your current development session or restart Visual Studio, all Visual Studio caches are immediately emptied, but begin to repopulate. The caches flushed include caches for the following features:

- Find all references
- Navigate To
- Add Using

In addition, caches used for internal Visual Studio operations are also cleared.

> **NOTE**
> The automatic feature suspension warning occurs only once on a per-solution basis, not on a per-session basis. This means that if you switch from Visual Basic to Visual C# (or vice-versa) and run into another low memory condition, you can possibly get another automatic feature suspension warning.

See also

- How to: Configure live code analysis scope for managed code
- Fundamentals of Garbage Collection
- Performance considerations for large solutions