

# Contents

[Code analysis documentation](#)

[Overview](#)

[Code analysis for managed code](#)

[Roslyn analyzers](#)

[Code analysis for C/C++](#)

[Measure code maintainability with code metrics](#)

[Quickstarts](#)

[Quickstart: Code analysis for C/C++](#)

[Tutorials](#)

[Analyze managed code for code defects](#)

[Analyze C/C++ code for defects](#)

[Demo sample](#)

[How-to guides](#)

[Code analysis for managed code](#)

[FxCop analyzers](#)

[FAQ](#)

[Install](#)

[Configure](#)

[Configuration options](#)

[Migrate from static code analysis](#)

[Analyzers FAQ](#)

[Install Roslyn analyzers](#)

[Use and configure Roslyn analyzers](#)

[Enable automatic code analysis](#)

[Enable full solution analysis](#)

[Automatic feature suspension](#)

[Run code analysis manually](#)

[Suppress warnings](#)

[Suppress code analysis warnings for generated code](#)

- [Customize the code analysis dictionary](#)
- [Anonymous methods and code analysis](#)
- [Create a work item for a managed code defect](#)
- [Code analysis for C/C++](#)
  - [Use the C++ Core Guidelines checkers](#)
  - [Set Code Analysis Properties](#)
  - [Use SAL Annotations to Reduce Code Defects](#)
    - [Understand SAL](#)
    - [Annotate Function Parameters and Return Values](#)
    - [Annotate Function Behavior](#)
    - [Annotate Structs and Classes](#)
    - [Annotate Locking Behavior](#)
    - [Specify When and Where an Annotation Applies](#)
    - [Intrinsic Functions](#)
    - [Best Practices and Examples \(SAL\)](#)
  - [Specify Additional Code Information by Using `\_Analysis\_assume`](#)
- [Rule sets](#)
  - [Rule sets for analyzer packages](#)
  - [Rule sets for managed code](#)
    - [Configure code analysis for an ASP.NET web app](#)
  - [Rule sets for C++ code](#)
  - [Create a custom rule set](#)
  - [Use the rule set editor](#)
- [Check-in policies](#)
  - [Implement Custom Code Analysis Check-in Policies for Managed Code](#)
  - [Enforce Maintainable Code](#)
  - [Synchronize Project Rule Sets with Team Check-in Policy](#)
  - [Version Compatibility](#)
- [Code metrics](#)
  - [Generate Code Metrics Data](#)
  - [Work with Code Metrics Data](#)
- [Reference](#)
  - [Rule sets](#)

All Rules rule set

Basic Correctness Rules rule set for managed code

Basic Design Guideline Rules rule set for managed code

Extended Correctness Rules rule set for managed code

Extended Design Guidelines Rules rule set for managed code

Globalization Rules rule set for managed code

Managed Minimum Rules rule set for managed code

Managed Recommended Rules rule set for managed code

Mixed Minimum Rules rule set

Mixed Recommended Rules rule set

Native Minimum Rules rule set

Native Recommended Rules rule set

Security Rules rule set for managed code

Code analysis for managed code warnings

Code Analysis warnings for Managed Code by CheckId

Cryptography warnings

CA5350: Do Not Use Weak Cryptographic Algorithms

CA5351 Do Not Use Broken Cryptographic Algorithms

Design warnings

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  
CA2210: Assemblies should have valid strong names

## Globalization warnings

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

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 GetLastErrorHandler 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

#### Maintainability warnings

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

#### Mobility warnings

CA1600: Do not use idle process priority  
CA1601: Do not use timers that prevent power state changes

#### Naming warnings

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

CA1800: Do not cast unnecessarily  
CA1802: Use Literals Where Appropriate  
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

CA1824: Mark assemblies with NeutralResourcesLanguageAttribute

## Portability warnings

CA1900: Value type fields should be portable

CA1901: P-Invoke declarations should be portable

CA1903: Use only API from targeted framework

## Reliability warnings

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

## Security warnings

CA2100: Review SQL queries for security vulnerabilities

CA2102: Catch non-CLSCCompliant 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

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

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

## Usage warnings

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

## Code Analysis Policy Errors

## C++ Core Guidelines Checker warnings

C26400

C26401

C26402

C26403

C26404

C26405

C26406

C26407

C26408

C26409

C26410

C26411

C26414

C26415

C26416

C26417

C26418

C26426

C26427

C26429

C26430

C26431

C26432

C26433

C26434

C26435

C26436

C26437

C26438

C26439

C26440

C26441

C26443

C26444

C26445

C26446

C26447

C26448

C26449

C26450

C26451

C26452

C26453

C26454

C26455

C26456

C26460

C26461

C26462

C26463

C26464

C26465

C26466

C26471

C26472

C26473

C26474

C26475

C26476

C26477

C26481

C26482

C26483

C26485

[C26486](#)

[C26487](#)

[C26488](#)

[C26489](#)

[C26490](#)

[C26491](#)

[C26492](#)

[C26493](#)

[C26494](#)

[C26495](#)

[C26496](#)

[C26497](#)

[C26498](#)

## [Code Analysis for C/C++ warnings](#)

[C1250](#)

[C1251](#)

[C1252](#)

[C1253](#)

[C1254](#)

[C1255](#)

[C1256](#)

[C1257](#)

[C6001](#)

[C6011](#)

[C6014](#)

[C6029](#)

[C6031](#)

[C6053](#)

[C6054](#)

[C6059](#)

[C6063](#)

[C6064](#)

C6066

C6067

C6101

C6102

C6103

C6200

C6201

C6211

C6214

C6215

C6216

C6217

C6219

C6220

C6221

C6225

C6226

C6230

C6235

C6236

C6237

C6239

C6240

C6242

C6244

C6246

C6248

C6250

C6255

C6258

C6259

C6260

C6262

C6263

C6268

C6269

C6270

C6271

C6272

C6273

C6274

C6276

C6277

C6278

C6279

C6280

C6281

C6282

C6283

C6284

C6285

C6286

C6287

C6288

C6289

C6290

C6291

C6292

C6293

C6294

C6295

C6296

C6297

C6298

C6299

C6302

C6303

C6305

C6306

C6308

C6310

C6312

C6313

C6314

C6315

C6316

C6317

C6318

C6319

C6320

C6322

C6323

C6324

C6326

C6328

C6329

C6330

C6331

C6332

C6333

C6334

C6335

C6336

C6340

C6381

C6383

C6384

C6385

C6386

C6387

C6388

C6400

C6401

C6411

C6412

C6500

C6501

C6503

C6504

C6505

C6506

C6508

C6509

C6510

C6511

C6513

C6514

C6515

C6516

C6517

C6518

C6522

C6525

C6527

C6530

C6540

C6551

C6552

C6701

C6702

C6703

C6704

C6705

C6706

C6707

C6993

C6995

C6997

C26100

C26101

C26105

C26110

C26111

C26112

C26115

C26116

C26117

C26130

C26135

C26138

C26140

C26160

C26165

C26166

C26167

C26800

C26810

C26811

C28020

C28021

C28022

C28023

C28024

C28039

C28103

C28104

C28105

C28106

C28107

C28108

C28109

C28112

C28113

C28125

C28137

C28138

C28159

C28160

C28163

C28164

C28182

C28183

C28193

C28194

C28195

C28196

C28197

C28198

C28199

C28202

C28203

C28204

C28205

C28206

C28207

C28208

C28209

C28210

C28211

C28212

C28213

C28214

C28215

C28216

C28217

C28218

C28219

C28220

C28221

C28222

C28223

C28224

C28225

C28226

C28227

C28228

C28229

C28230

C28231

C28232

C28233

C28234

C28235

C28236

C28237

C28238

C28239

C28240

C28241

C28243

C28244

C28245

C28246

C28250

C28251

C28252

C28253

C28254

C28262

C28263

C28267

C28272

C28273

C28275

C28278

C28279

C28280

C28282

C28283

C28284

C28285

C28286

C28287

C28288

C28289

C28290

[C28291](#)

[C28300](#)

[C28301](#)

[C28302](#)

[C28303](#)

[C28304](#)

[C28305](#)

[C28306](#)

[C28307](#)

[C28308](#)

[C28309](#)

[C28310](#)

[C28311](#)

[C28312](#)

[C28350](#)

[C28351](#)

## [Code Analysis Application Errors](#)

[CA0001](#)

[CA0051](#)

[CA0052](#)

[CA0053](#)

[CA0054](#)

[CA0055](#)

[CA0056](#)

[CA0057](#)

[CA0058](#)

[CA0059](#)

[CA0060](#)

[CA0061](#)

[CA0062](#)

[CA0063](#)

[CA0064](#)

[CA0065](#)

[CA0066](#)

[CA0067](#)

[CA0068](#)

[CA0069](#)

[CA0070](#)

[CA0501](#)

[CA0502](#)

[CA0503](#)

[CA0504](#)

[CA0505](#)

[FxCopCmd Errors](#)

Visual Studio provides several different tools to analyze and improve code quality.

## [Analyze managed code quality](#)

## [Analyze C and C++ code quality](#)

## [Measure code maintainability with code metrics](#)

---

## Reference

### [Managed code analysis warnings](#)

### [C++ code analysis warnings](#)

### [Rule sets](#)

# Overview of static code analysis for managed code in Visual Studio

3/1/2019 • 2 minutes to read • [Edit Online](#)

Visual Studio can perform code analysis of managed code in two ways: with *FxCop* static analysis of managed assemblies, and with the more modern *Roslyn analyzers*. This topic covers FxCop static code analysis. To learn more about analyzing code by using code analyzers, see [Overview of Roslyn 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 Microsoft .NET Framework 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

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 [Roslyn 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 **Enable Code Analysis on Build** on the project's 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 or 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 and want to become productive right away, you can *baseline* the analysis state of your project. From the **Analyze** menu, select **Run Code Analysis 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 Roslyn analyzers](#)
- [Using Rule Sets to Group Code Analysis Rules](#)
- [How to: Enable and Disable Automatic Code Analysis](#)

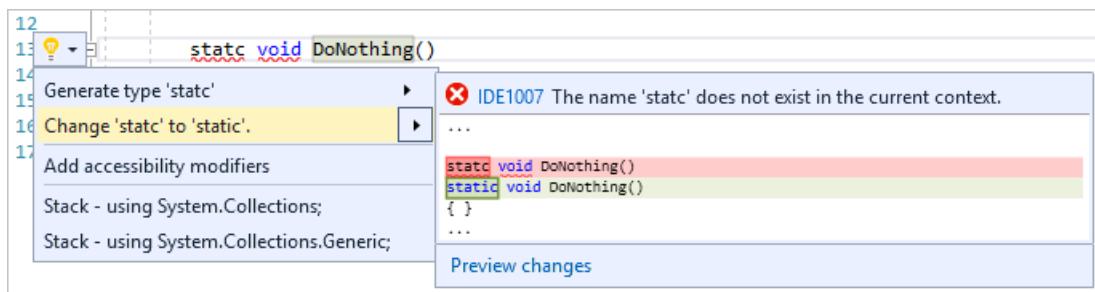
# Overview of .NET Compiler Platform analyzers

4/8/2019 • 3 minutes to read • [Edit Online](#)

.NET Compiler Platform ("Roslyn") analyzers analyze your code for style, quality and maintainability, design, and other issues. Visual Studio includes a built-in set of analyzers that analyze your C# or Visual Basic code as you type. You configure preferences for these built-in analyzers on the [text editor Options](#) page or in an [.editorconfig file](#). You can install additional analyzers as a Visual Studio extension or a NuGet package.

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

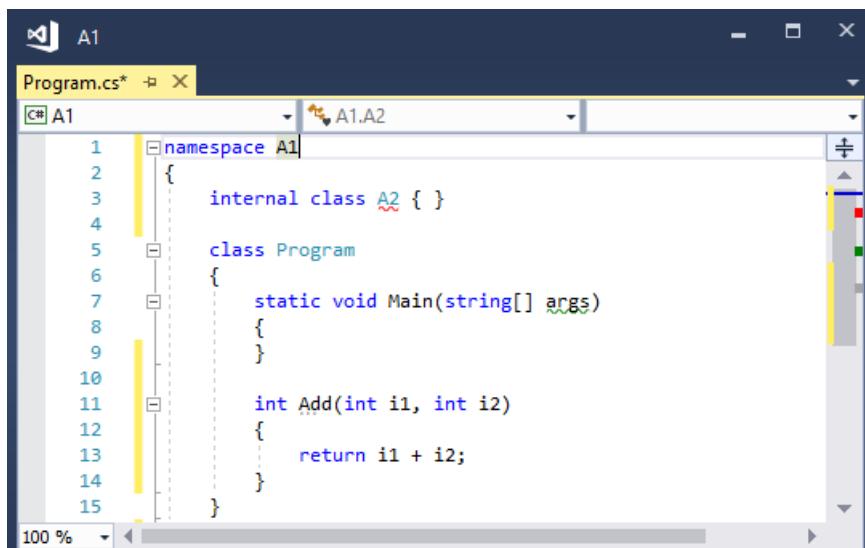
Many analyzer rules, or *diagnostics*, have one or more associated *code fixes* that you can apply to correct the problem. The analyzer diagnostics that are built into Visual Studio each have an associated code fix. Code fixes are shown in the light bulb icon menu along with other types of [Quick Actions](#).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



## Roslyn analyzers vs. static code analysis

.NET Compiler Platform ("Roslyn") analyzers will eventually replace [static code analysis](#) for managed code. Many of the static code analysis rules have already been rewritten as Roslyn analyzer diagnostics.

Like static code analysis rule violations, Roslyn analyzer violations appear in **Error List**. In addition, Roslyn analyzer violations also show up in the code editor as *squiggles* under the offending code. The color of the squiggly depends on the [severity setting](#) of the rule. The following screenshot shows three violations—one red, one green, and one gray:



Roslyn analyzers analyze code at build time, like static code analysis if it's enabled, but also live as you type. If you enable [full solution analysis](#), Roslyn analyzers also provide design-time analysis of code files that aren't open

in the editor.

#### TIP

Build-time errors and warnings from Roslyn analyzers are shown only if the analyzers are installed as a NuGet package.

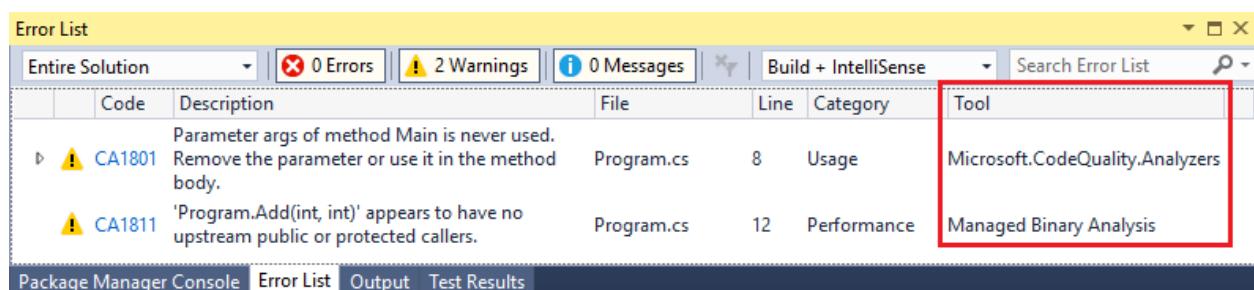
Not only do Roslyn analyzers report the same types of problems that static code analysis does, but they make it easy for you to fix one, or all, occurrences of the violation in your file or project. These actions are called *code fixes*. Code fixes are IDE-specific; in Visual Studio, they are implemented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

#### NOTE

The following UI options apply only to static code analysis:

- The **Analyze > Run Code Analysis** menu option.
- The **Enable Code Analysis on Build** and **Suppress results from generated code** checkboxes on the **Code Analysis** tab of a project's property pages (these options have no effect on Roslyn analyzers).

To differentiate between violations from Roslyn analyzers and static code 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.CodeAnalysis.Analyzers**, the violation comes from a Roslyn analyzer. Otherwise, the violation originates from static code analysis.



#### TIP

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

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

## NuGet package versus VSIX extension

.NET Compiler Platform analyzers can be installed per-project via a NuGet package, or Visual Studio-wide as a Visual Studio extension. There are some key behavior differences between these two methods of [installing analyzers](#).

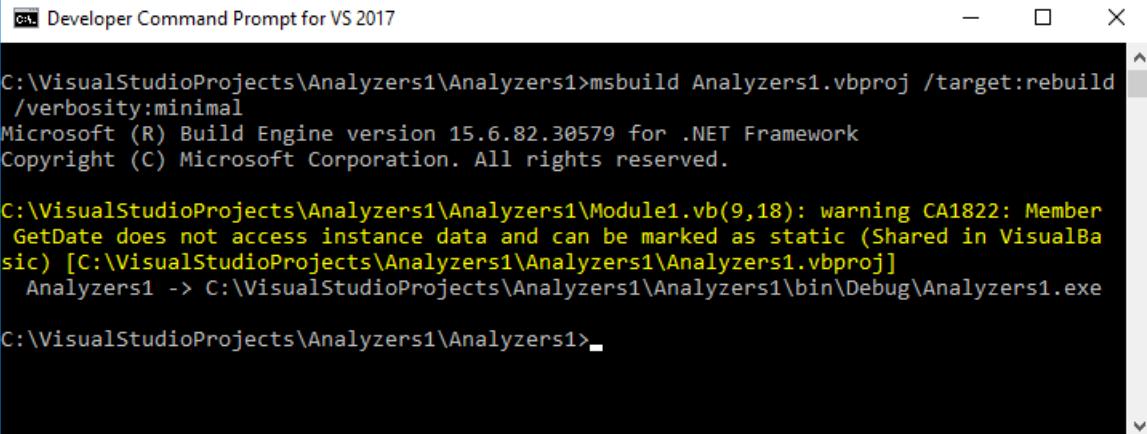
### Scope

If you install analyzers as a Visual Studio extension, they apply at the solution level, 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 screenshot shows the command-line build output from building a project that contains an analyzer rule violation:



A screenshot of a 'Developer Command Prompt for VS 2017' window. The title bar says 'Developer Command Prompt for VS 2017'. The console output shows the following:

```
C:\VisualStudioProjects\Analyzers1\Analyzers1>msbuild Analyzers1.vbproj /target:rebuild /verbosity:minimal
Microsoft (R) Build Engine version 15.6.82.30579 for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

C:\VisualStudioProjects\Analyzers1\Analyzers1\Module1.vb(9,18): warning CA1822: Member 'GetDate' does not access instance data and can be marked as static (Shared in VisualBasic) [C:\VisualStudioProjects\Analyzers1\Analyzers1\Analyzers1.vbproj]
  Analyzers1 -> C:\VisualStudioProjects\Analyzers1\Analyzers1\bin\Debug\Analyzers1.exe

C:\VisualStudioProjects\Analyzers1\Analyzers1>
```

### Rule severity

You cannot set 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 Roslyn analyzers in Visual Studio](#)

[Use Roslyn analyzers in Visual Studio](#)

## See also

- [Analyzers FAQ](#)
- [Write your own Roslyn analyzer](#)
- [.NET Compiler Platform SDK](#)

# Code analysis for C/C++ overview

2/8/2019 • 2 minutes to read • [Edit Online](#)

The C/C++ Code Analysis tool provides information about possible defects in your C/C++ source code. Common coding errors reported by the tool include buffer overruns, uninitialized memory, null pointer dereferences, and memory and resource leaks. The tool can also run checks against the [C++ Core Guidelines](#).

## IDE (integrated development environment) integration

The code analysis tool is fully integrated within the Visual Studio IDE.

During the build process, any warnings generated for the source code appear in the Error List. You can navigate to source code that caused the warning, and you can view additional information about the cause and possible solutions of the issue.

## Command line support

You can also use the analysis tool from the command line, as shown in the following example:

```
C:\>ccl /analyze Sample.cpp
```

**Visual Studio 2017 version 15.7 and later** You can run the tool from the command line with any build system including CMake.

## #pragma support

You can use the `#pragma` directive to treat warnings as errors; enable or disable warnings, and suppress warnings for individual lines of code. For more information, see [How to: Set Code Analysis Properties for C/C++ Projects](#).

## Annotation support

Annotations improve the accuracy of the code analysis. Annotations provide additional information about pre- and post- conditions on function parameters and return types. For more information, see [How to: Specify Additional Code Information by Using \\_\\_analysis\\_assume](#)

## Run analysis tool as part of check-in policy

You might want to require that all source code check-ins satisfy certain policies. In particular, you want to make sure that analysis was run as a step of the most recent local build. For more information about enabling a code analysis check-in policy, see [Creating and Using Code Analysis Check-In Policies](#)

## Team Build integration

You can use the integrated features of the build system to run code analysis tool as a step of the Team Foundation Server build process. For more information, see [Azure Pipelines](#).

## See also

- [Quickstart: Code analysis for C/C++](#)
- [Walkthrough: Analyze C/C++ Code for Defects](#)

- [Code Analysis for C/C++ Warnings](#)
- [Use the C++ Core Guidelines checkers](#)
- [C++ Core Guidelines Checker Reference](#)
- [Use Rule Sets to Specify the C++ Rules to Run](#)
- [Analyze Driver Quality by Using Code Analysis Tools](#)
- [Code Analysis for Drivers Warnings](#)

# Code metrics values

2/8/2019 • 3 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.
- **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 will require more tests to achieve good code coverage and will be less maintainable.
- **Depth of Inheritance** - Indicates the number of class definitions that extend to the root of the class hierarchy. The deeper the hierarchy the more difficult it might be to understand where particular methods and fields are defined or/and redefined.
- **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.
- **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 very 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. Metrics results for an anonymous method that is 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.

For more information about how Code Metrics treats anonymous methods, see [Anonymous Methods and Code Analysis](#).

## 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](#)

# Quickstart: Code analysis for C/C++

2/8/2019 • 4 minutes to read • [Edit Online](#)

You can improve the quality of your application by running code analysis regularly on C or C++ code. This can help you find common problems, violations of good programming practice, or defects that are difficult to discover through testing. Code analysis warnings differ from compiler errors and warnings because code analysis searches for specific code patterns that are valid but could still create issues for you or other people who use your code.

## Configure rule sets for a project

1. In **Solution Explorer**, open the shortcut menu for the project name and then choose **Properties**.
2. The following steps are optional:
  - a. In the **Configuration** and **Platform** lists, choose the build configuration and target platform.
  - b. By default, code analysis does not report warnings from code that is automatically generated by external tools. 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.

3. To run code analysis every time the project is built using the selected configuration, select the **Enable Code Analysis for C/C++ on Build** check box. You can also run code analysis manually by opening the **Analyze** menu and then choosing **Run Code Analysis on ProjectName**.
4. In the **Run this rule set** list, do one of the following:
  - Choose the rule set that you want to use.
  - Choose <Browse...> to specify an existing custom rule set that is not in the list.
  - Define a [custom rule set](#).

### Standard C/C++ Rule Sets

Visual Studio includes two standard sets of rules for native code:

RULE SET	DESCRIPTION
Microsoft Native Minimum Recommended Rules	This rule set focuses on the most critical problems in your native code, including potential security holes and application crashes. You should include this rule set in any custom rule set you create for your native projects.
Microsoft Native Recommended Rules	This rule set covers a broad range of problems. It includes all the rules in Microsoft Native Minimum Recommended Rules.

## Run code analysis

On the Code analysis page of the project properties pages, you can configure code analysis to run each time you build your project. You can also run code analysis manually.

To run code analysis on a solution:

- On the **Build** menu, choose **Run Code Analysis on Solution**.

To run code analysis on a project:

1. In Solution Explorer, choose the name of the project.
2. On the **Build** menu, choose **Run Code Analysis on Project Name**.

The project or solution is compiled and code analysis runs. Results appear in the Error List.

## Analyze and resolve code analysis warnings

To analyze a specific warning, choose the title of the warning in the Error List. The warning expands to display additional information about the issue. When possible, code analysis displays the line numbers and analysis logic that led to the warning. For detailed information about the warning, including possible solutions to the issue, choose the warning ID to display its corresponding online help topic.

When you select a warning, the line of code that caused the warning is highlighted in the Visual Studio code editor.

After you understand the problem, you can resolve it in your code. Then, rerun code analysis to make sure that the warning no longer appears in the Error List, and that your fix has not raised any new warnings.

## Suppress code analysis warnings

There are times when you might decide not to fix a code analysis warning. You might decide that resolving the warning requires too much recoding in relation to the probability that the issue will arise in any real-world implementation of your code. Or you might believe that the analysis that is used in the warning is inappropriate for the particular context. You can suppress individual warnings so that they no longer appear in the Error List.

To suppress a warning:

1. If the detailed information is not displayed, choose the title of the warning to expand it.
2. Choose the **Actions** link at the bottom of the warning.
3. Choose **Suppress Message** and then choose **In Source**.

Suppressing a message inserts `#pragma warning (disable:[warning ID])` that suppresses the warning for the line of code.

## Create work items for code analysis warnings

You can use the work item tracking feature to log bugs from within Visual Studio. To use this feature, you must connect to an instance of Team Foundation Server.

### To create a work item for one or more C/C++ code warnings

1. In the Error List, expand and select the warnings
2. On the shortcut menu for the warnings, choose **Create Work Item**, and then choose the work item type.
3. Visual Studio creates a single work item for the selected warnings and displays the work item in a document window of the IDE.
4. Add any additional information, and then choose **Save Work Item**.

# Search and filter code analysis results

You can search long lists of warning messages and you can filter warnings in multi-project solutions.

- **To filter warnings by title or warning id:** Enter the keyword in the search box.
- **To filter warnings by severity:** By default, code analysis messages are assigned a severity of **Warning**. You can assign the severity of one or more messages as **Error** in a custom rule set. On the **Severity** column of the **Error List**, choose the drop-down arrow and then the filter icon. Choose **Warning** or **Error** to display only the messages that are assigned the respective severity. Choose **Select All** to display all messages.

## See also

[Code analysis for C/C++](#)

# Walkthrough: Analyzing managed code for code defects

2/8/2019 • 4 minutes to read • [Edit Online](#)

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

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

## Create a class library

### To create a class library

1. On the **File** menu, choose **New > Project**.
2. In the **New Project** dialog box, expand **Installed > Visual C#**, and then choose **Windows Desktop**.
3. Choose the **Class Library (.NET Framework)** template.
4. In the **Name** text box, type **CodeAnalysisManagedDemo** and then click **OK**.
5. After the project is created, open the *Class1.cs* file.
6. 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; } }
    }
}
```

7. Save the *Class1.cs* file.

## Analyze the project

### To analyze a managed 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 checked.
5. From the **Run this rule set** 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

### To correct code analysis rule violations

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:** Microsoft.Design: 'demo' should be marked with the `CLSCompliantAttribute`, and its value should be true.

a. Add the code `using System;` to the *AssemblyInfo.cs* file.

b. Next, add the code `[assembly: CLSCompliant(true)]` to the end of the *AssemblyInfo.cs* file.

**CA1032: Implement standard exception constructors:** Microsoft.Design: Add the following constructor to this class: `public demo(String)`

a. Add the constructor `public demo (String s) : base(s) { }` to the class `demo`.

**CA1032: Implement standard exception constructors:** Microsoft.Design: Add the following constructor to this class: `public demo(String, Exception)`

a. Add the constructor `public demo (String s, Exception e) : base(s, e) { }` to the class `demo`.

**CA1032: Implement standard exception constructors:** Microsoft.Design: Add the following constructor to this class: `protected demo(SerializationInfo, StreamingContext)`

a. Add the code `using System.Runtime.Serialization;` to the beginning of the *Class1.cs* file.

b. Next, add the constructor

`protected demo (SerializationInfo info, StreamingContext context) : base(info, context) { }` to the class `demo`.

**CA1032: Implement standard exception constructors:** Microsoft.Design: Add the following constructor to this class: `public demo()`

a. Add the constructor `public demo () : base() { }` to the class `demo`.

**CA1709: Identifiers should be cased correctly:** Microsoft.Naming: Correct the casing of namespace name 'testCode' by changing it to 'TestCode'.

a. Change the casing of the namespace `testCode` to `TestCode`.

**CA1709: Identifiers should be cased correctly:** Microsoft.Naming: Correct the casing of type name 'demo' by changing it to 'Demo'.

a. Change the name of the member to `Demo`.

**CA1709: Identifiers should be cased correctly:** Microsoft.Naming: Correct the casing of member name 'item' by changing it to 'Item'.

a. Change the name of the member to `Item`.

**CA1710: Identifiers should have correct suffix:** Microsoft.Naming: Rename 'testCode.demo' to end in

'Exception'.

- a. Change the name of the class and its constructors to `DemoException`.

[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. In the **Key file name**, type `TestKey`.

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

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

[CA2237: Mark ISerializable types with SerializableAttribute](#): Microsoft.Usage: Add a [Serializable] attribute to type 'demo' as this type implements ISerializable.

- a. Add the `[Serializable ()]` attribute to the class `demo`.

After you complete 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](#)

# Walkthrough: Analyzing C/C++ Code for Defects

3/11/2019 • 3 minutes to read • [Edit Online](#)

This walkthrough demonstrates how to analyze C/C++ code for potential code defects by using the code analysis tool for C/C++ code.

- Run code analysis on native code.
- Analyze code defect warnings.
- Treat warning as an error.
- Annotate source code to improve code defect analysis.

## Prerequisites

- A copy of the [Demo Sample](#).
- Basic understanding of C/C++.

### To run code defect analysis on native code

1. Open the Demo solution in Visual Studio.

The Demo solution now populates **Solution Explorer**.

2. On the **Build** menu, click **Rebuild Solution**.

The solution builds without any errors or warnings.

3. In **Solution Explorer**, select the CodeDefects project.

4. On the **Project** menu, click **Properties**.

The **CodeDefects Property Pages** dialog box is displayed.

5. Click **Code Analysis**.

6. Click the **Enable Code Analysis for C/C++ on Build** check box.

7. Rebuild the CodeDefects project.

Code analysis warnings are displayed in the **Error List**.

### To analyze code defect warnings

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

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

2. In the **Error List**, double-click the following warning:

warning C6230: Implicit cast between semantically different types: using HRESULT in a Boolean context.

The code editor displays the line that caused the warning in the function `bool ProcessDomain()`. This warning indicates that a HRESULT is being used in an 'if' statement where a Boolean result is expected.

3. Correct this warning by using the SUCCEEDED macro. Your code should resemble the following code:

```
if (SUCCEEDED (ReadUserAccount()) )
```

4. In the **Error List**, double-click the following warning:

warning C6282: Incorrect operator: assignment to constant in test context. Was == intended?

5. Correct this warning by testing for equality. Your code should look similar to the following code:

```
if ((len == ACCOUNT_DOMAIN_LEN) || (g_userAccount[len] != '\\'))
```

### To treat warning as an error

1. In the Bug.cpp file, add the following `#pragma` statement to the beginning of the file to treat the warning C6001 as an error:

```
#pragma warning (error: 6001)
```

2. Rebuild the CodeDefects project.

In the **Error List**, C6001 now appears as an error.

3. Correct the remaining two C6001 errors in the **Error List** by initializing `i` and `j` to 0.
4. Rebuild the CodeDefects project.

The project builds without any warnings or errors.

### To correct the source code annotation warnings in annotation.c

1. In Solution Explorer, select the Annotations project.

2. On the **Project** menu, click **Properties**.

The **Annotations Property Pages** dialog box is displayed.

3. Click **Code Analysis**.

4. Select the **Enable Code Analysis for C/C++ on Build** check box.

5. Rebuild the Annotations project.

6. In the **Error List**, double-click the following warning:

warning C6011: Dereferencing NULL pointer 'newNode'.

This warning indicates failure by the caller to check the return value. In this case, a call to **AllocateNode** might return a NULL value (see the annotations.h header file for function declaration for AllocateNode).

7. Open the annotations.cpp file.

8. To correct this warning, use an 'if' statement to test the return value. Your code should resemble the following code:

```
if (NULL != newNode)
{
    newNode->data = value;
    newNode->next = 0;
    node->next = newNode;
}
```

9. Rebuild the Annotations project.

The project builds without any warnings or errors.

## To use source code annotation

1. Annotate formal parameters and return value of the function `AddTail` by using the Pre and Post conditions as shown in this example:

```
[returnvalue:SA_Post (Null=SA_Maybe)] LinkedList* AddTail  
(  
[SA_Pre(Null=SA_Maybe)] LinkedList* node,  
int value  
)
```

2. Rebuild Annotations project.
3. In the **Error List**, double-click the following warning:

warning C6011: Dereferencing NULL pointer 'node'.

This warning indicates that the node passed into the function might be null, and indicates the line number where the warning was raised.

4. To correct this warning, use an 'if' statement to test the return value. Your code should resemble the following code:

```
...  
LinkedList *newNode = NULL;  
if (NULL == node)  
{  
    return NULL;  
    ...  
}
```

5. Rebuild Annotations project.

The project builds without any warnings or errors.

## See also

[Walkthrough: Analyzing Managed Code for Code Defects](#) [Code analysis for C/C++](#)

# Sample C++ project for code analysis

2/8/2019 • 4 minutes to read • [Edit Online](#)

This following procedures show you how to create the sample for [Walkthrough: Analyze C/C++ code for defects](#).

The procedures create:

- A Visual Studio solution named CppDemo.
- A static library project named CodeDefects.
- A static library project named Annotations.

The procedures also provide the code for the header and .cpp files for the static libraries.

## Create the CppDemo solution and the CodeDefects project

1. Click the **File** menu, point to **New**, and then click **New Project**.
2. In the **Project types** tree list, if Visual C++ is not your default language in VS expand **Other Languages**.
3. Expand **Visual C++**, and then click **General**.
4. In **Templates**, click **Empty Project**.
5. In the **Name** text box, type **CodeDefects**.
6. Select the **Create directory for solution** check box.
7. In the **Solution Name** text box, type **CppDemo**.

## Configure the CodeDefects project as a static library

1. In Solution Explorer, right-click **CodeDefects** and then click **Properties**.
2. Expand **Configuration Properties** and then click **General**.
3. In the **General** list, select the text in the column next to **Target Extension**, and then type **.lib**.
4. In **Project Defaults**, click the column next to **Configuration Type**, and then click **Static Lib (.lib)**.

## Add the header and source file to the CodeDefects project

1. In Solution Explorer, expand **CodeDefects**, right-click **Header Files**, click **Add**, and then click **New Item**.
2. In the **Add New Item** dialog box, click **Code**, and then click **Header File (.h)**.
3. In the **Name** box, type **Bug.h** and then click **Add**.
4. Copy the following code and paste it into the *Bug.h* file in the Visual Studio editor.

```
#include <windows.h>

//
//These 3 functions are consumed by the sample
// but are not defined. This project cannot be linked!
//

bool CheckDomain( LPCSTR );
HRESULT ReadUserAccount();

//
//These constants define the common sizes of the
// user account information throughout the program
//

const int USER_ACCOUNT_LEN = 256;
const int ACCOUNT_DOMAIN_LEN = 128;
```

5. In Solution Explorer, right-click **Source Files**, point to **New**, and then click **New Item**.
6. In the **Add New Item** dialog box, click **C++ File (.cpp)**
7. In the **Name** box, type **Bug.cpp** and then click **Add**.
8. Copy the following code and paste it into the *Bug.cpp* file in the Visual Studio editor.

```

#include <stdlib.h>
#include "Bug.h"

// the user account
TCHAR g_userAccount[USER_ACCOUNT_LEN] = "";
int len = 0;

bool ProcessDomain()
{
    TCHAR* domain = new TCHAR[ACCOUNT_DOMAIN_LEN];
    // ReadUserAccount gets a 'domain\user' input from
    // the user into the global 'g_userAccount'
    if (ReadUserAccount() )
    {

        // Copies part of the string prior to the '\'
        // character onto the 'domain' buffer
        for( len = 0 ; (len < ACCOUNT_DOMAIN_LEN) && (g_userAccount[len] != '\0') ; len++ )
        {
            if ( g_userAccount[len] == '\\')
            {
                // Stops copying on the domain and user separator ('\'')
                break;
            }
            domain[len] = g_userAccount[len];
        }
        if((len== ACCOUNT_DOMAIN_LEN) || (g_userAccount[len] != '\\'))
        {
            // '\' was not found. Invalid domain\user string.
            delete [] domain;
            return false;
        }
        else
        {
            domain[len]='\0';
        }
        // Process domain string
        bool result = CheckDomain( domain );

        delete[] domain;
        return result;
    }
    return false;
}

int path_dependent(int n)
{
    int i;
    int j;
    if (n == 0)
        i = 1;
    else
        j = 1;
    return i+j;
}

```

9. Click the **File** menu, and then click **Save All**.

## Add the Annotations project and configure it as a static library

1. In Solution Explorer, click **CppDemo**, point to **Add**, and then click **New Project**.
2. In the **Add New Project** dialog box, expand Visual C++, click **General**, and then click **Empty Project**.
3. In the **Name** text box, type **Annotations**, and then click **Add**.

4. In Solution Explorer, right-click **Annotations** and then click **Properties**.
5. Expand **Configuration Properties** and then click **General**.
6. In the **General** list, select the text in the column next to **Target Extension**, and then type **.lib**.
7. In **Project Defaults**, click the column next to **Configuration Type**, and then click **Static Lib (.lib)**.

## Add the header file and source file to the Annotations project

1. In Solution Explorer, expand **Annotations**, right-click **Header Files**, click **Add**, and then click **New Item**.
2. In the **Add New Item** dialog box, click **Header File (.h)**.
3. In the **Name** box, type **annotations.h** and then click **Add**.
4. Copy the following code and paste it into the *annotations.h* file in the Visual Studio editor.

```
#include <CodeAnalysis/SourceAnnotations.h>

struct LinkedList
{
    struct LinkedList* next;
    int data;
};

typedef struct LinkedList LinkedList;

[returnvalue:SA_Post( Null=SA_Maybe )] LinkedList* AllocateNode();
```

5. In Solution Explorer, right-click **Source Files**, point to **New**, and then click **New Item**.
6. In the **Add New Item** dialog box, click **Code** and then click **C++ File (.cpp)**
7. In the **Name** box, type **annotations.cpp** and then click **Add**.
8. Copy the following code and paste it into the *annotations.cpp* file in the Visual Studio editor.

```
#include <CodeAnalysis/SourceAnnotations.h>
#include <windows.h>
#include <stdlib.h>
#include "annotations.h"

LinkedList* AddTail( LinkedList *node, int value )
{
    LinkedList *newNode = NULL;

    // finds the last node
    while ( node->next != NULL )
    {
        node = node->next;
    }

    // appends the new node
    newNode = AllocateNode();
    newNode->data = value;
    newNode->next = 0;
    node->next = newNode;

    return newNode;
}
```

9. Click the **File** menu, and then click **Save All**.

# Frequently asked questions about FxCop and FxCop analyzers

4/8/2019 • 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?

No. When you select **Analyze > Run Code Analysis**, it executes static code analysis or legacy FxCop. **Run Code Analysis** has no effect on Roslyn-based analyzers, including the Roslyn-based FxCop analyzers.

## 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**. This is equivalent to selecting **Analyze > Run Code Analysis** in Visual Studio.

## 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, static code analysis will execute after each build.

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

## See also

- [Overview of .NET Compiler Platform analyzers](#)
- [Get started with analyzers](#)
- [Install FxCop analyzers](#)

# Install FxCop analyzers in Visual Studio

3/29/2019 • 2 minutes to read • [Edit Online](#)

Microsoft created a set of analyzers, called [Microsoft.CodeAnalysis.FxCopAnalyzers](#), that contains the most important "FxCop" rules from static code analysis, converted to Roslyn analyzers. These analyzers check your code for security, performance, and design issues, among others.

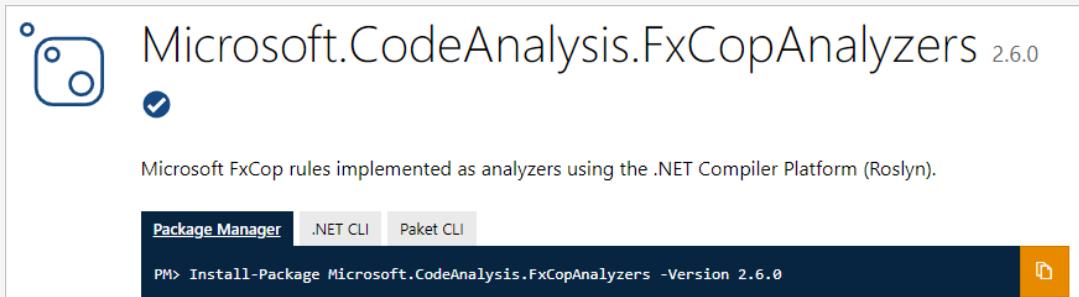
You can install these FxCop analyzers either as a NuGet package or as a VSIX extension to Visual Studio. To learn about the pros and cons of each, see [NuGet package vs. VSIX extension](#).

## To install FxCop analyzers as a NuGet package

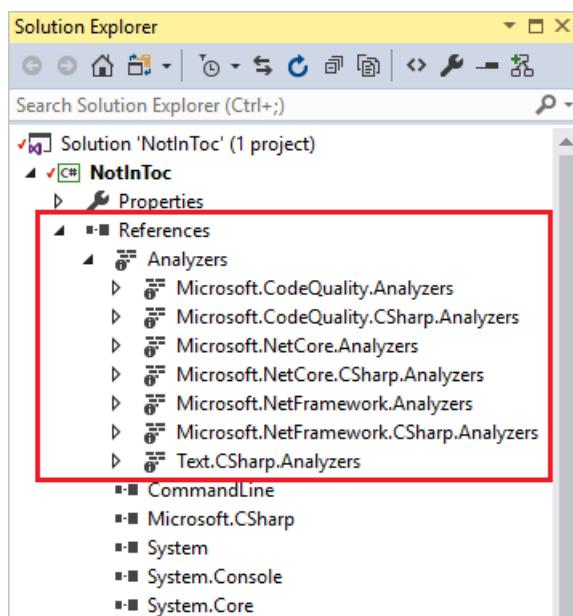
1. [Determine which analyzer package version](#) to install, based on your version of Visual Studio.
2. Install the package in Visual Studio, using either the [Package Manager Console](#) or the [Package Manager UI](#).

### 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 clipboard.



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



## 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 2017 version 15.8 and later	<a href="#">2.9.1</a>
Visual Studio 2017 version 15.5 to 15.7	<a href="#">2.6.3</a>
Visual Studio 2017 version 15.3 to 15.4	<a href="#">2.3.0-beta1</a>
Visual Studio 2017 version 15.0 to 15.2	<a href="#">2.0.0-beta2</a>
Visual Studio 2015 update 2 and 3	<a href="#">1.2.0-beta2</a>
Visual Studio 2015 Update 1	<a href="#">1.1.0</a>
Visual Studio 2015 RTW	<a href="#">1.0.1</a>

## To install FxCop analyzers as a 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](#).

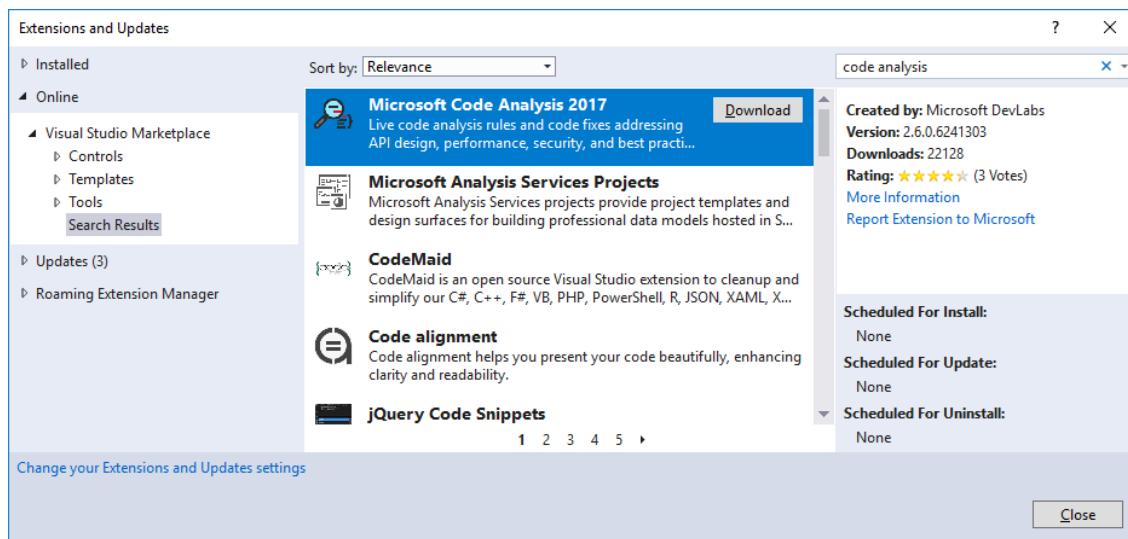
1. In Visual Studio, select **Extensions > Manage Extensions**.

The **Manage Extensions** dialog box opens.

**NOTE**

Alternatively, download the extension directly from [Visual Studio Marketplace](#).

1. Expand **Online** in the left pane, and then select **Visual Studio Marketplace**.
2. In the search box, type "code analysis", and look for the **Microsoft Code Analysis 2017** extension.



3. Select **Download**.

The extension is downloaded.

4. 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.



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

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

7. 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 Roslyn analyzers in Visual Studio](#)
- [Use Roslyn analyzers in Visual Studio](#)

- Migrate from FxCop to Roslyn analyzers

# Configure FxCop analyzers

3/12/2019 • 2 minutes to read • [Edit Online](#)

The [FxCop analyzers](#) consist of the most important "FxCop" rules from static code analysis, converted to Roslyn analyzers. You can configure FxCop code analyzers in two ways:

- With a [rule set](#), which lets you enable or disable rule and set the severity for individual rule violations.
- Starting in version 2.6.3 of the [Microsoft.CodeAnalysis.FxCopAnalyzers](#) NuGet package, through an [.editorconfig file](#). The [configurable options](#) let you refine which parts of your codebase to analyze.

## TIP

For information about the differences between FxCop static code analysis and FxCop analyzers, see [FxCop analyzers FAQ](#).

## FxCop analyzer rule sets

One way to configure FxCop analyzers is by using an XML *rule set*. A rule set is a grouping of code analysis rules that identify targeted issues and specific conditions. Rule sets let you enable or disable rule and set the severity for individual rule violations.

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

- design
- documentation
- maintainability
- naming
- performance
- reliability
- security
- usage

For more information, see [Rule sets for Roslyn analyzers](#).

## EditorConfig file

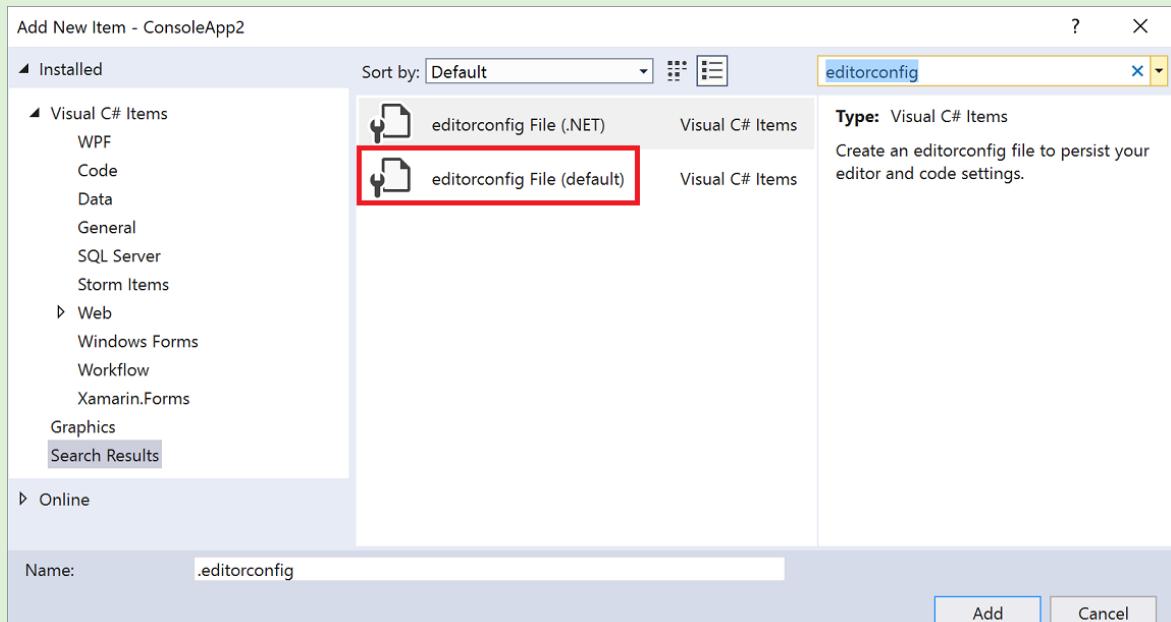
You can configure analyzer rules by adding key-value pairs to an [.editorconfig](#) file. A configuration file can be [specific to a project](#) or it can be [shared](#) between two or more projects.

### Per-project configuration

To enable .editorconfig-based analyzer configuration for a specific project, add an [.editorconfig](#) file to the project's root directory.

## TIP

You can 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**.



Currently there is no hierarchical support for "combining" `.editorconfig` files that exist at different directory levels, for example, the solution and project level.

## Shared configuration

You can share an `.editorconfig` file for 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.CodeAnalysis.FxCopAnalyzers.props"
Condition="Exists('..\packages\Microsoft.CodeAnalysis.FxCopAnalyzers.2.6.3\build\Microsoft.CodeAnalysis.FxCopAnalyzers.props')"/>
...
```

4. Reload the project.

#### NOTE

You cannot configure legacy FxCop rules (static code analysis FxCop) by using an .editorconfig file.

## Option scopes

Each 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
<code>dotnet_code_quality.OptionName = OptionValue</code>	<code>dotnet_code_quality.api_surface = public</code>

### 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
<code>dotnet_code_quality.RuleCategory.OptionName = OptionValue</code>	<code>dotnet_code_quality.Naming.api_surface = public</code>

### Specific rule

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

SYNTAX	EXAMPLE
<code>dotnet_code_quality.RuleId.OptionName = OptionValue</code>	<code>dotnet_code_quality.CA1040.api_surface = public</code>

## See also

- [Analyzer configuration](#)
- [FxCop analyzers](#)

# Configuration options for FxCop analyzers

3/12/2019 • 2 minutes to read • [Edit Online](#)

This page lists the available configuration options, their allowable values, and the configurable rules for each option.

## api\_surface

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Which part of the API surface to analyze	<input type="checkbox"/> public <input type="checkbox"/> internal or <input type="checkbox"/> friend <input type="checkbox"/> private <input type="checkbox"/> all	<input type="checkbox"/> public	<a href="#">CA1000</a> <a href="#">CA1003</a> <a href="#">CA1008</a> <a href="#">CA1010</a> <a href="#">CA1012</a> <a href="#">CA1024</a> <a href="#">CA1027</a> <a href="#">CA1028</a> <a href="#">CA1030</a> <a href="#">CA1036</a> <a href="#">CA1040</a> <a href="#">CA1041</a> <a href="#">CA1043</a> <a href="#">CA1044</a> <a href="#">CA1051</a> <a href="#">CA1052</a> <a href="#">CA1054</a> <a href="#">CA1055</a> <a href="#">CA1056</a> <a href="#">CA1058</a> <a href="#">CA1063</a> <a href="#">CA1708</a> <a href="#">CA1710</a> <a href="#">CA1711</a> <a href="#">CA1714</a> <a href="#">CA1715</a> <a href="#">CA1716</a> <a href="#">CA1717</a> <a href="#">CA1720</a> <a href="#">CA1721</a> <a href="#">CA1725</a> <a href="#">CA1802</a> <a href="#">CA1815</a> <a href="#">CA1819</a> <a href="#">CA2217</a> <a href="#">CA2225</a> <a href="#">CA2226</a> <a href="#">CA2231</a> <a href="#">CA2234</a>

## exclude\_async\_void\_methods

DESCRIPTION	ALLOWABLE VALUES	DEFAULT VALUE	CONFIGURABLE RULES
Whether to ignore asynchronous methods that don't return a value	<code>true</code> <code>false</code>	<code>false</code>	<a href="#">CA2007</a>

#### 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 <a href="#">type parameters</a> from the rule, for example, <code>s</code> in <code>Collection&lt;S&gt;</code>	<code>true</code> <code>false</code>	<code>false</code>	<a href="#">CA1715</a>

#### 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 <a href="#">OutputKind</a> enumeration  Separate multiple values with a comma (,)	All output kinds	<a href="#">CA2007</a>

# Analyzers FAQ

3/13/2019 • 2 minutes to read • [Edit Online](#)

This page contains answers to some frequently asked questions about Roslyn analyzers in Visual Studio.

## Roslyn analyzers versus .editorconfig

**Q:** Should I use Roslyn analyzers or .editorconfig for code style?

**A:** Roslyn analyzers 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 Roslyn analyzers that are built into Visual Studio. EditorConfig files can also be used to configure some third-party 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 are mutually exclusive ways to configure analyzers. They can coexist. [Rule sets](#) let you enable and disable rules and set their severity. EditorConfig files offer other ways to configure rules. For FxCop analyzers, .editorconfig files let you [define which types of code to analyze](#). For the 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 third-party analyzers are configured through the use of text files marked as [additional files](#) for the C# and VB compilers.

### NOTE

EditorConfig files cannot be used to configure static code analysis rules, whereas rule sets can.

## Analyzers in CI builds

**Q:** Do analyzers 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.

## Analyzers versus static code analysis

**Q:** What's the difference between analyzers and static code analysis?

**A:** Analyzers analyze source code in real time and during compilation, whereas static code analysis analyzes binary files after build has completed. For more information, see [Roslyn analyzers vs. static code analysis](#) and [FxCop analyzers FAQ](#).

## See also

- [Analyzers overview](#)
- [.NET coding convention settings for EditorConfig](#)

# Install .NET Compiler Platform analyzers

3/14/2019 • 2 minutes to read • [Edit Online](#)

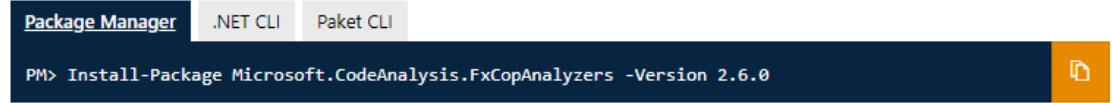
Visual Studio includes a core set of .NET Compiler Platform (*Roslyn*) analyzers. These analyzers are always on. You can install additional analyzers either as NuGet packages, or as Visual Studio extensions in VSIX files.

## To install NuGet analyzer packages

1. Find the analyzer package you want to install on [www.nuget.org](http://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.
2. Install the package in Visual Studio, using either the [Package Manager Console](#) or the [Package Manager UI](#).

### NOTE

The [www.nuget.org](http://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 Roslyn analyzers in Visual Studio](#)

## See also

- [Overview of Roslyn analyzers in Visual Studio](#)
- [Install FxCop analyzers](#)

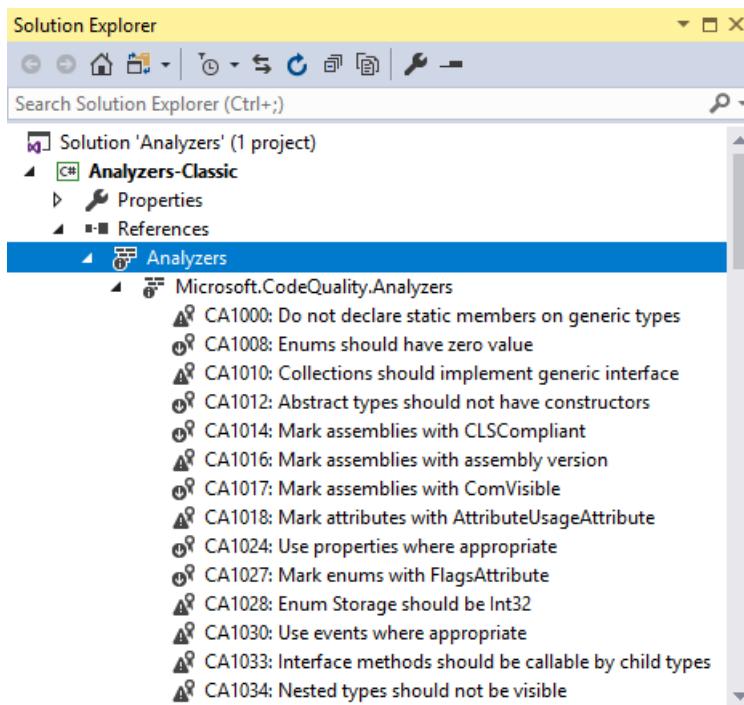
# Use Roslyn analyzers

3/27/2019 • 7 minutes to read • [Edit Online](#)

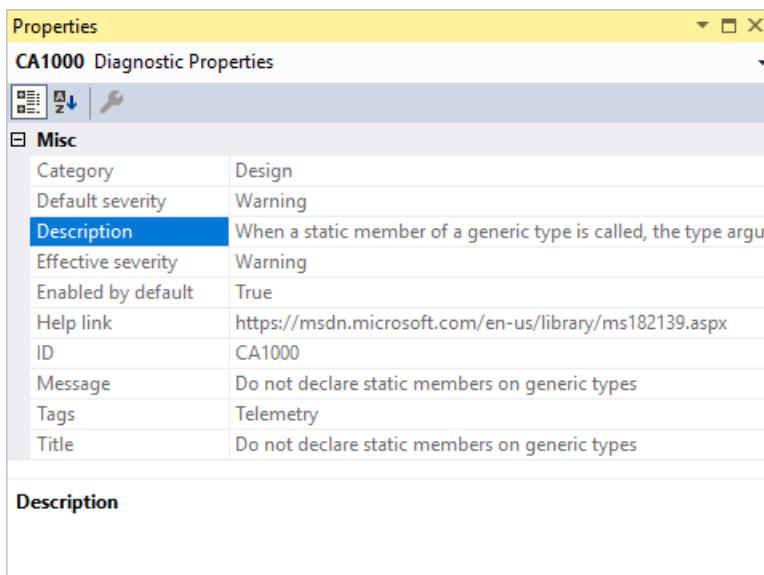
.NET Compiler Platform ("Roslyn") analyzer rules, or *diagnostics*, analyze your C# or Visual Basic code as you type. Each diagnostic has a default severity and suppression state that can be overwritten for your project. This article covers setting rule severity, using rule sets, and suppressing violations.

## 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.



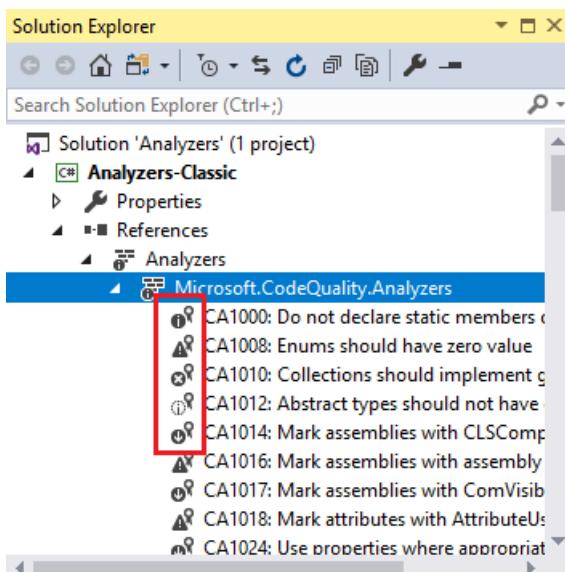
You can view the properties of a diagnostic, including its description and default severity, in the **Properties** window. To view the properties, right-click on the rule and select **Properties**, or select the rule and then press **Alt+Enter**.



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 "i" in a circle indicates a **severity** of **Info**
- the "!" in a triangle indicates a **severity** of **Warning**
- the "x" in a circle indicates a **severity** of **Error**
- 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 sets

A **rule set** is an XML file that stores the severity and suppression state for individual diagnostics.

### NOTE

Rule sets can include rules from both static (binary) code analysis and Roslyn analyzers.

To edit the active rule set in the rule set editor, right-click on the **References > Analyzers** node in **Solution Explorer** and select **Open Active Rule Set**. 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.

To change the active rule set for a project, navigate to the **Code Analysis** tab of a project's properties. Select the rule set from the list under **Run this rule set**. To open the rule set, select **Open**.

#### 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 configure the rules within the rule set in the Visual Studio rule set editor UI.

## Rule severity

You can configure the severity of analyzer rules, or *diagnostics*, if you [install the analyzers](#) as a NuGet package.

The following table shows the severity options for diagnostics:

SEVERITY	BUILD-TIME BEHAVIOR	EDITOR BEHAVIOR
Error	Violations appear as <i>Errors</i> in the <b>Error List</b> and in command-line build output, and cause builds to fail.	Offending code is underlined with a red squiggly, and marked by a small red box in the scroll bar.
Warning	Violations appear as <i>Warnings</i> in the <b>Error List</b> and in command-line build output, but do not cause builds to fail.	Offending code is underlined with a green squiggly, and marked by a small green box in the scroll bar.
Info	Violations appear as <i>Messages</i> in the <b>Error List</b> , and not at all in command-line build output.	Offending code is underlined with a gray squiggly, and marked by a small gray box in the scroll bar.
Hidden	Non-visible to user.	Non-visible to user. The diagnostic is reported to the IDE diagnostic engine, however.
None	Suppressed completely.	Suppressed completely.

In addition, you can "reset" a rule's severity by setting it to **Default**. Each diagnostic has a default severity that can be seen in the **Properties** window.

The following screenshot shows three different diagnostic violations in the code editor, with three different severities. Notice the color of the squiggly, as well as the small box in the scroll bar on the right.

The screenshot shows a Visual Studio code editor window for a C# project named 'A1'. The file 'Program.cs' is open, containing the following code:

```

1  namespace A1
2  {
3      internal class A2 { }
4
5      class Program
6      {
7          static void Main(string[] args)
8          {
9          }
10
11         int Add(int i1, int i2)
12         {
13             return i1 + i2;
14         }
15     }

```

Three violations are highlighted with red squiggly lines under the code:

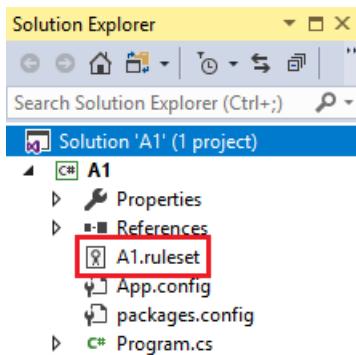
- Line 3: 'internal class A2 { }' - CA1812
- Line 7: 'static void Main(string[] args)' - CA1801
- Line 11: 'int Add(int i1, int i2)' - CA1822

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

Error List				
Code	Description	File	Line	Category
CA1812	A2 is an internal class that is apparently never instantiated. If so, remove the code from the assembly. If this class is intended to contain only static members, make it static (Shared in Visual Basic).	Program.cs	3	Performance Microsoft.CodeAnalysis.Analyzers
CA1801	Parameter args of method Main is never used. Remove the parameter or use it in the method body.	Program.cs	7	Usage Microsoft.CodeAnalysis.Analyzers
CA1822	Member Add does not access instance data and can be marked as static (Shared in VisualBasic)	Program.cs	11	Performance Microsoft.CodeAnalysis.Analyzers

Package Manager Console Error List Output Test Results

You can change the severity of a rule from **Solution Explorer**, or within the `<projectname>.ruleset` file that is added to the solution after you change the severity of a rule in **Solution Explorer**.



### Set rule severity from Solution Explorer

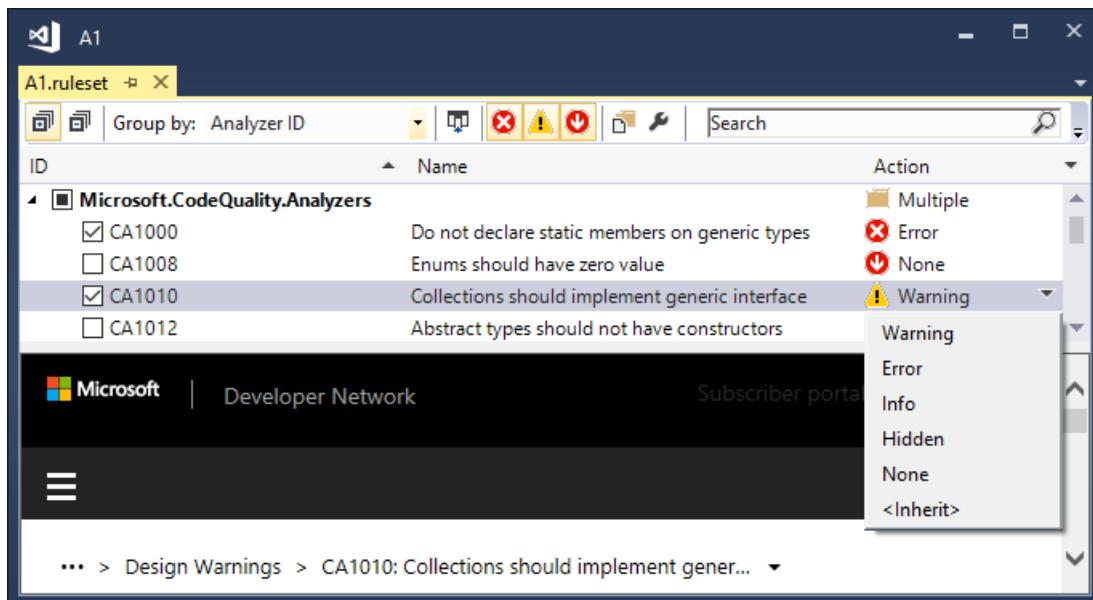
1. In **Solution Explorer**, expand **References > Analyzers (Dependencies > Analyzers for .NET Core projects)**.
2. Expand the assembly that contains the rule you want to set severity for.
3. Right-click on the rule and select **Set Rule Set Severity**. In the fly-out menu, select one of the severity 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 **rule set** file by double-clicking it in **Solution Explorer**, selecting **Open Active Rule Set** on the right-click menu of the **Analyzers** node, or by selecting **Open** on the **Code Analysis** property page for the project.
2. Browse to the rule by expanding its containing assembly.

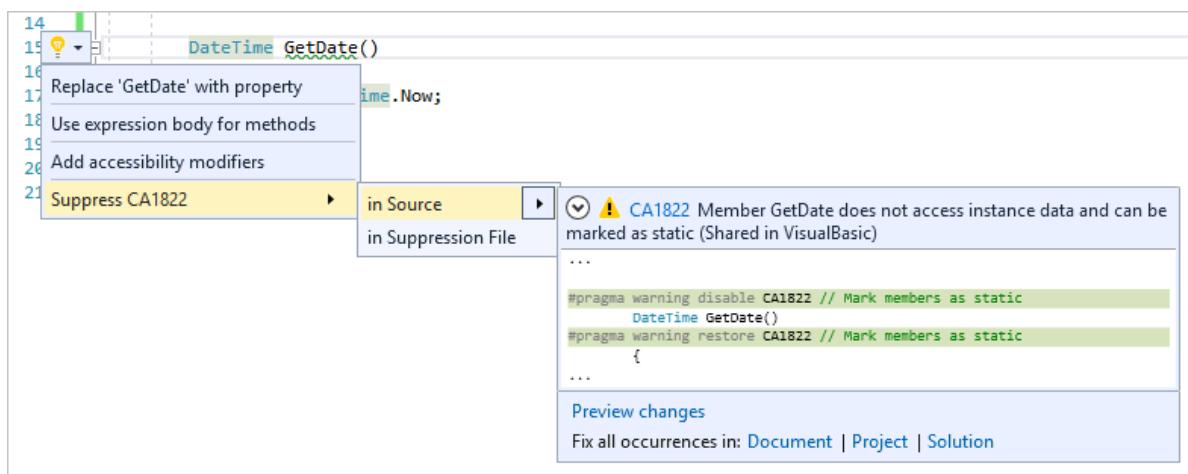
3. In the **Action** column, select the value to open a drop-down list, and select the desired severity from the list.



## Suppress violations

There are multiple ways to suppress rule violations:

- To suppress all current violations, select **Analyze > Run Code Analysis and Suppress Active Issues** on the menu bar. This is sometimes referred to as "baselining".
- To suppress a diagnostic from **Solution Explorer**, set its severity to **None**.
- To suppress a diagnostic from the rule set editor, uncheck the box next to its name, or set **Action** to **None**.
- To suppress a diagnostic from the code editor, place the cursor in the line of code with the violation and press **Ctrl+.** to open the **Quick Actions** menu. Select **Suppress CAxxxx > In Source** or **Suppress CAxxxx > In Suppression File**.

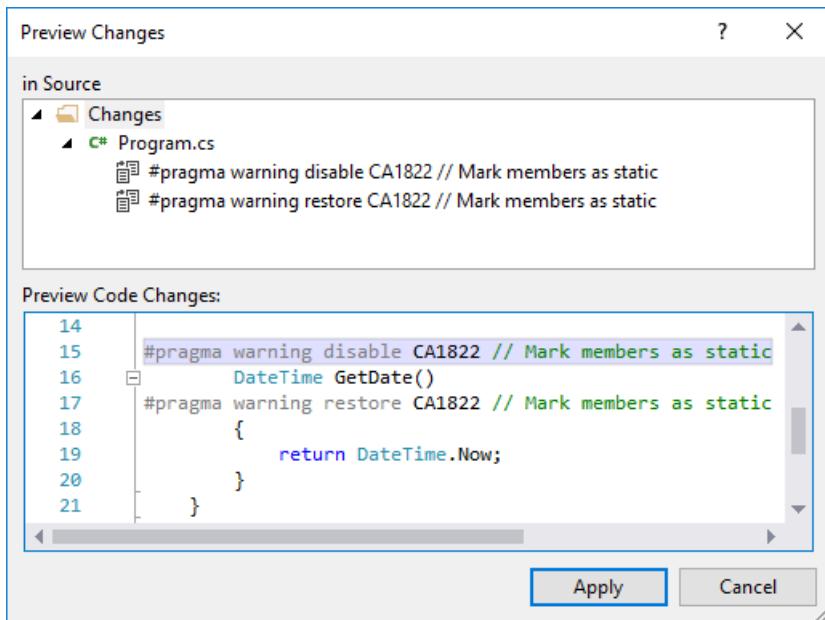


- To suppress a diagnostic from the **Error List**, see [Suppress violations from the Error List](#).

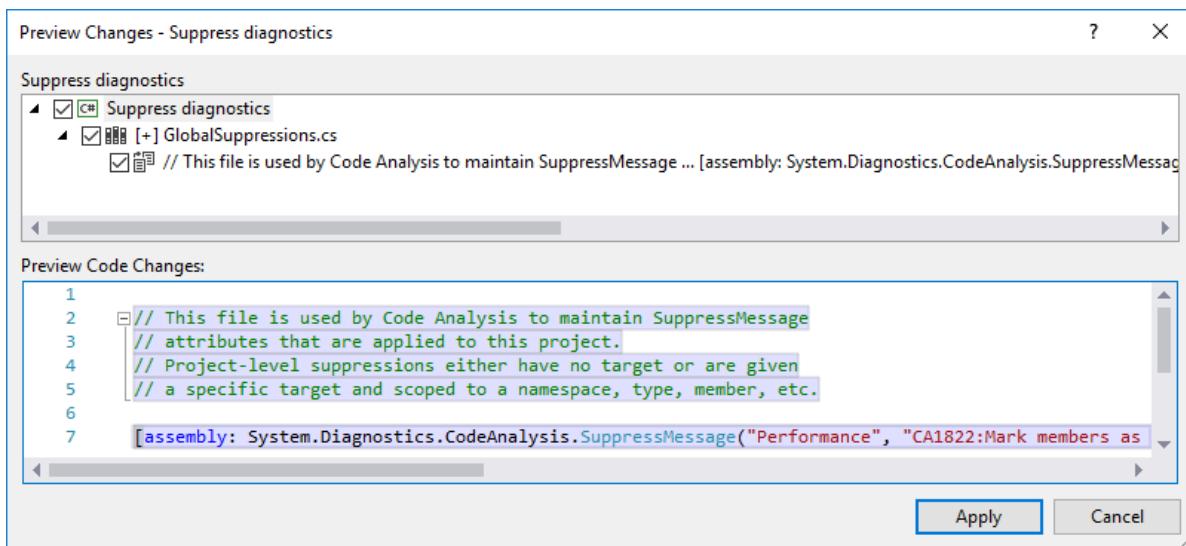
### Suppress violations from the Error List

You can suppress one or many diagnostics from the **Error List** by selecting the ones you want to suppress, and then right-clicking and selecting **Suppress > In Source** or **Suppress > In Suppression File**.

- If you select **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.

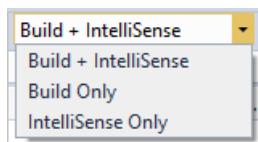


- If you select **In Suppression File**, the **Preview Changes** dialog opens and shows a preview of the [SuppressMessageAttribute](#) attribute that's added to the global suppressions file.



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

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**. However, diagnostics from live analysis, or IntelliSense, are always up-to-date with current sources, and can be suppressed from the **Error List**. If the suppression option is disabled in the right-click, or context, menu, it's likely because you have one or more build diagnostics in your selection. To exclude the 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.



#### NOTE

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:

```
<PackageReference Include="Microsoft.CodeAnalysis.FxCopAnalyzers" Version="2.6.0" PrivateAssets="all" />
```

## 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.

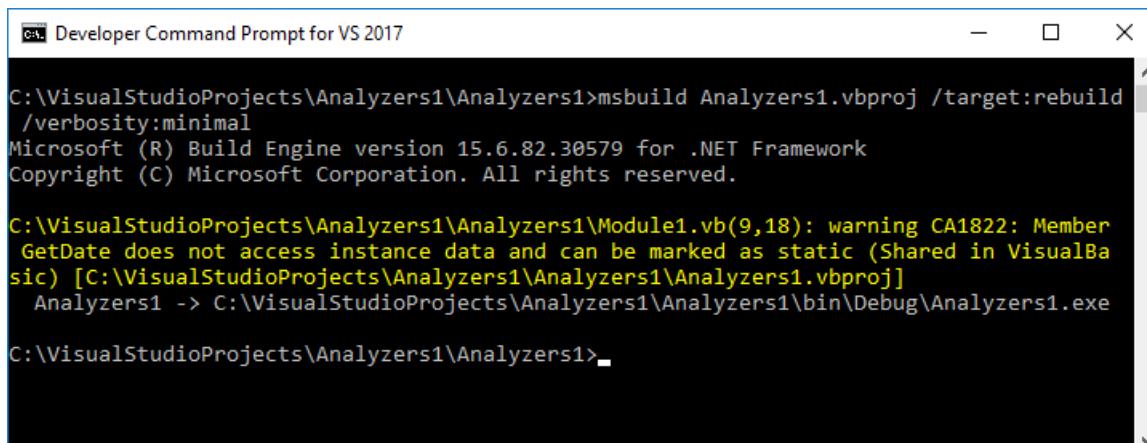
#### TIP

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



The screenshot shows a terminal window titled "Developer Command Prompt for VS 2017". The command entered is "msbuild Analyzers1.vbproj /target:rebuild /verbosity:minimal". The output shows the build engine version and copyright information, followed by a warning message: "C:\VisualStudioProjects\Analyzers1\Analyzers1\Module1.vb(9,18): warning CA1822: Member 'GetDate' does not access instance data and can be marked as static (Shared in VisualBasic) [C:\VisualStudioProjects\Analyzers1\Analyzers1\Analyzers1.vbproj]". The build continues successfully, producing the executable "Analyzers1.exe".

## See also

- [Overview of Roslyn analyzers in Visual Studio](#)
- [Submit a Roslyn analyzer bug](#)

- [Use rule sets](#)
- [Suppress code analysis warnings](#)

# How to: Enable and disable automatic code analysis for managed code

2/8/2019 • 2 minutes to read • [Edit Online](#)

You can configure (static) code analysis to run after each build of a managed code project. You can set different code analysis properties for each build configuration, for example, debug and release.

This article applies only to static code analysis and not live code analysis using [Roslyn code analyzers](#).

## To enable or disable automatic 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.

### TIP

Newer project types such as .NET Core and .NET Standard applications don't have a **Code Analysis** tab. Static code analysis is not available for these project types, but you can still get live code analysis using [Roslyn code analyzers](#). To suppress warnings from Roslyn code analyzers, see the note at the end of this article.

3. Specify the build type in **Configuration** and the target platform in **Platform**.
4. To enable or disable automatic code analysis, select or clear the **Enable Code Analysis on Build** check box.

### NOTE

The **Enable Code Analysis on Build** check box only affects static code analysis. It doesn't affect [Roslyn code analyzers](#), which always execute at build if you installed them as a NuGet package. 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](#).

# How to: Enable and disable full solution analysis for managed code

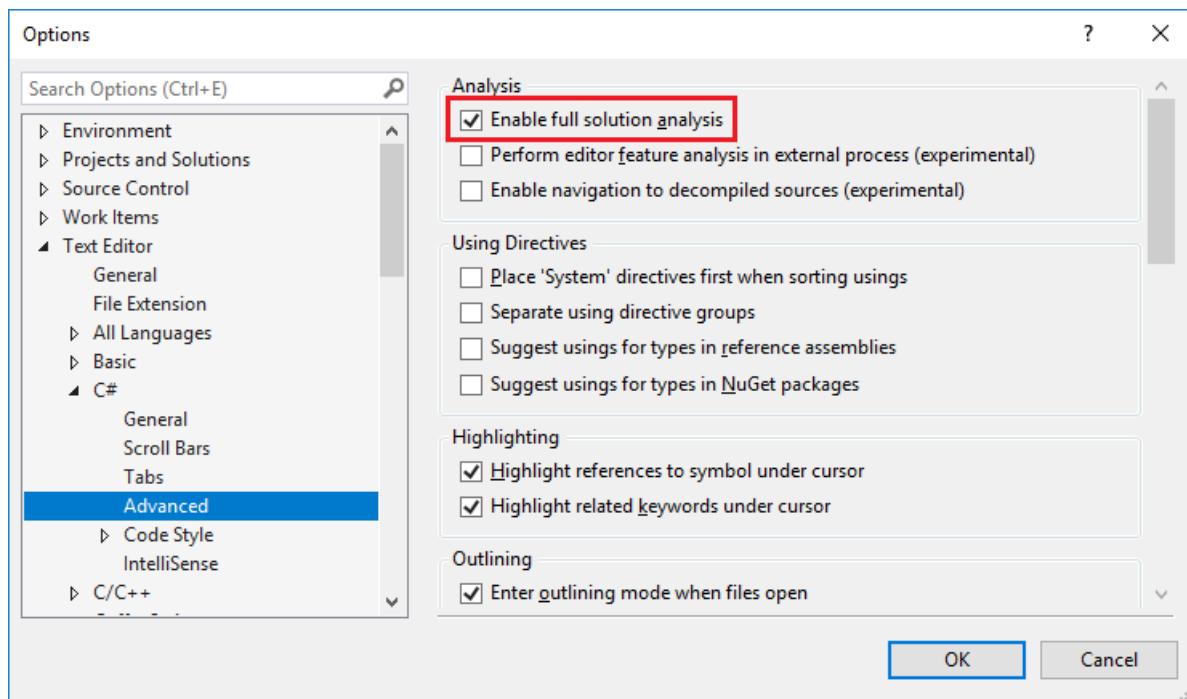
2/8/2019 • 2 minutes to read • [Edit Online](#)

*Full solution analysis* is a Visual Studio feature that enables you to see code analysis issues only in open Visual C# or Visual Basic files in your solution, or also in code files that are closed. By default, full solution analysis is *enabled* for Visual Basic, and *disabled* for Visual C#.

It can be useful to see all issues in all files, but it can also be distracting. It slows Visual Studio down if your solution is very large or has many files. To limit the number of issues shown and improve Visual Studio performance, you can disable full solution analysis. You can easily reenable this feature if necessary.

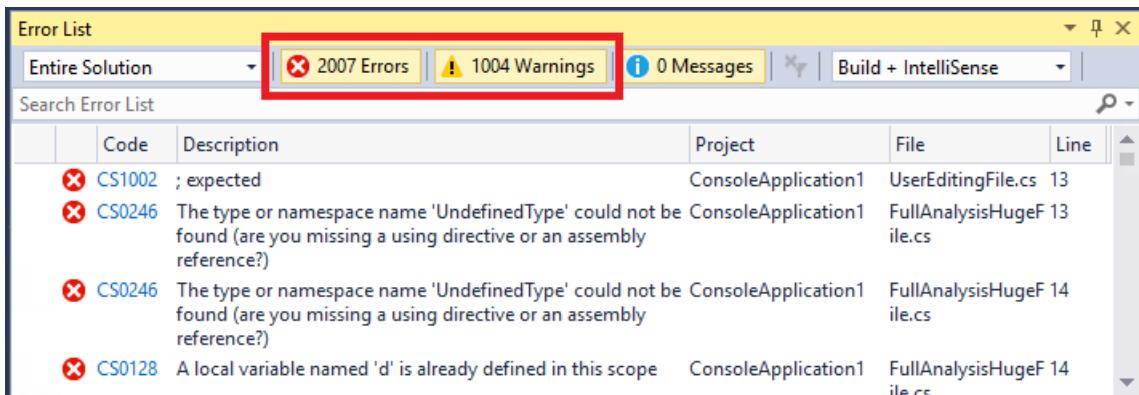
## To toggle full solution analysis

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 **Enable full solution analysis** check box to enable full solution analysis, or clear the box to disable it. Choose **OK** when you're done.

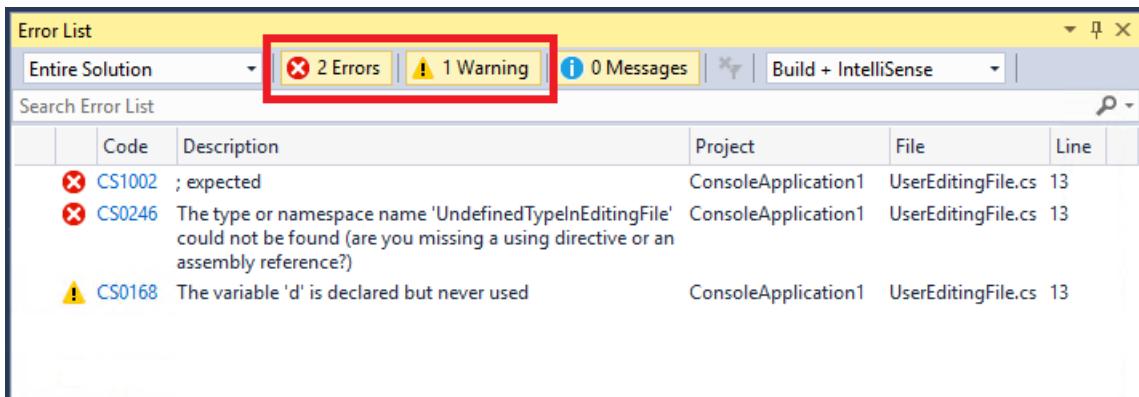


## Results of enabling and disabling full solution analysis

In the following screenshot, you can see the results when full solution analysis is enabled. All errors and code analysis issues in *all* of the files in the solution appear, regardless of whether the files are open or not.

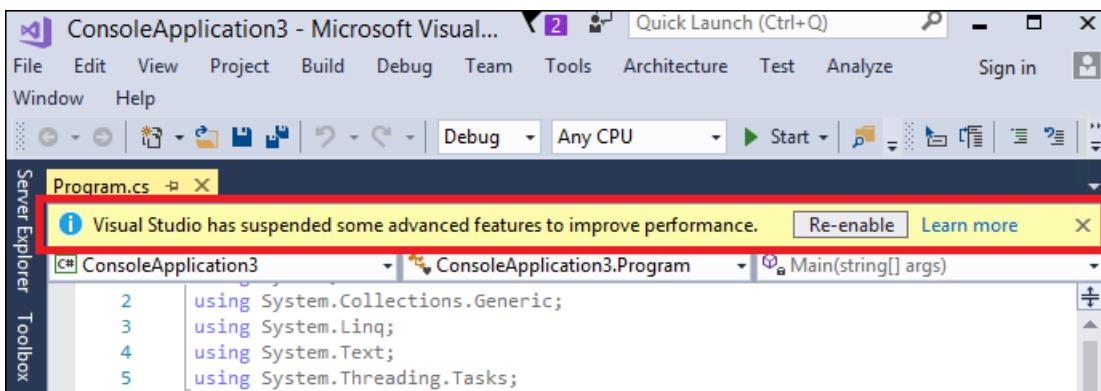


The following screenshot shows the results from the same solution after disabling full solution analysis. Only errors and code analysis issues in open solution files appear in the **Error List**.



## Automatically disable full solution analysis

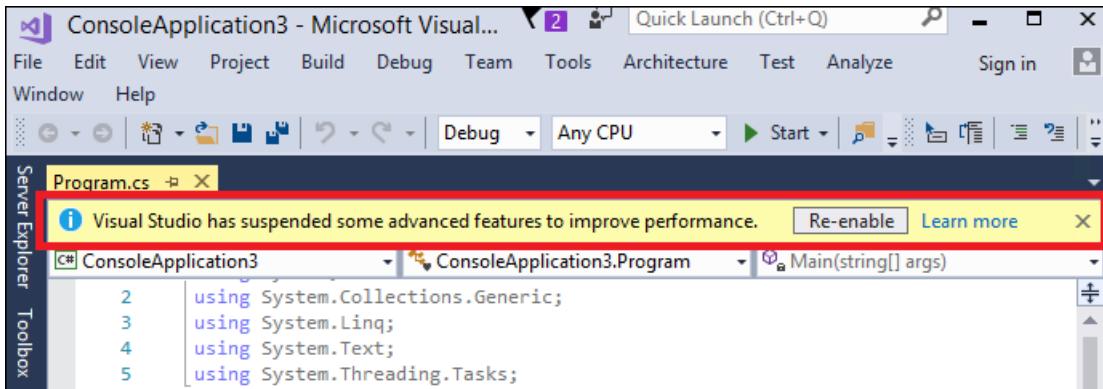
If Visual Studio detects that 200 MB or less of system memory is available to it, it automatically disables full solution analysis (and some other features) if it's enabled. If this occurs, an alert appears informing you that Visual Studio has disabled some features. A button lets you reenable full solution analysis if you want.



# Automatic feature suspension

2/8/2019 • 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:

- Full solution analysis for Visual C# and Visual Basic is disabled.
- [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](#).

## Full solution analysis suspended

By default, full solution analysis is enabled for Visual Basic and disabled for Visual C#. However, in a low memory condition, full solution analysis is automatically disabled for both Visual Basic and Visual C#, regardless of their settings in the Options dialog box. However, you can re-enable full solution analysis by choosing the **Re-enable** button in the info bar when it appears, by selecting the **Enable full solution analysis** check box in the Options dialog, or by restarting Visual Studio. The Options dialog box always shows the current full solution analysis settings. For more information, see [How to: Enable and Disable Full Solution Analysis](#).

## 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: Enable and Disable Full Solution Analysis](#)
- [Fundamentals of Garbage Collection](#)
- [Performance considerations for large solutions](#)

# How to: Run Code Analysis Manually for Managed Code

2/8/2019 • 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. In **Solution Explorer**, click the project.
2. On the **Analyze** menu, click **Run Code Analysis on *Project Name***.

# Suppress code analysis warnings

3/1/2019 • 6 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.

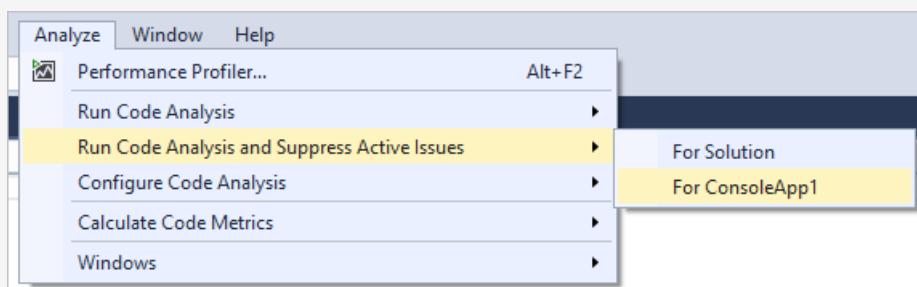
In C++/CLI, use the macros CA\_SUPPRESS\_MESSAGE or CA\_GLOBAL\_SUPPRESS\_MESSAGE in the header file to add the attribute.

## 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 or Visual Studio 2019, you might suddenly be faced with a large number of code analysis warnings. These warnings are coming from [Roslyn analyzers](#). 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**.



## 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`
  - `resource`
  - `type`
  - `member`
  - `namespace` - This scope suppresses warnings against the namespace itself. It does not suppress warnings against types within the namespace.
  - `namespaceanddescendants` - (New for Visual Studio 2019) This scope suppresses warnings in a namespace and all its descendant symbols. The `namespaceanddescendants` value is only valid for Roslyn analyzers, and is ignored by binary, FxCop-based static 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.

## 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}");
    }
}
```

## 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.

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.

## 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.

## See also

- [Scope](#)
- [System.Diagnostics.CodeAnalysis](#)
- [Use Roslyn analyzers](#)

# How to: Suppress Code Analysis Warnings for Generated Code

2/8/2019 • 2 minutes to read • [Edit Online](#)

Managed code compilers often generate code that is added to a project to facilitate rapid code development. In addition, developers often use third-party tools to help develop applications quickly. These tools also generate code that is added to the project.

You might want to see the rule violations that Code Analysis discovers in generated code. However, you might not want to see them if you cannot view and maintain the code that contains the violation.

The **Suppress results from generated code** check box on the Code Analysis property page of a project enables you to select whether you want to see 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. Click **Code Analysis**.
3. Select the **Suppress results from generated code** check box.

# How to: Customize the Code Analysis Dictionary

2/8/2019 • 5 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 Framework 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 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>
    <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:
2. 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.
3. 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**.
4. In **Solution Explorer**, right-click the **CustomDictionary.xml** file name and click **Properties**.
5. From the **Build Action** list, select **CodeAnalysisDictionary**.
6. From the **Copy to Output Directory** list, select **Do not copy**.

# Anonymous Methods and Code Analysis

2/8/2019 • 2 minutes to read • [Edit Online](#)

An *anonymous method* is a method that has no name. Anonymous methods are most frequently used to pass a code block as a delegate parameter. This article explains how code analysis handles warnings and metrics for anonymous methods.

## Anonymous Methods Declared In a Member

Warnings and metrics 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.

For example, in the following class, any warnings that are found in the declaration of **anonymousMethod** should be raised against **Method1** and not **Method2**.

```
Delegate Function ADelegate(ByVal value As Integer) As Boolean

Class AClass
    Sub Method1()
        Dim anonymousMethod As ADelegate = Function(ByVal value As Integer) value > 5
        Method2(anonymousMethod)
    End Sub
    Sub Method2(ByVal anonymousMethod As ADelegate)
        anonymousMethod(10)
    End Sub
End Class
```

```
delegate void Delegate();

class Class
{
    void Method1()
    {
        Delegate anonymousMethod = delegate()
        {
            Console.WriteLine("");
        }
        Method2(anonymousMethod);
    }

    void Method2(Delegate anonymousMethod)
    {
        anonymousMethod();
    }
}
```

## Inline Anonymous Methods

Warnings and metrics for an anonymous method that's declared as an inline assignment to a field are associated with the constructor. If the field is declared as `static` (`Shared` in Visual Basic), the warnings and metrics are associated with the class constructor. Otherwise, they're associated with the instance constructor.

For example, in the following class, any warnings that are found in the declaration of **anonymousMethod1** will be

raised against the implicitly generated default constructor of **Class**. Warnings that are found in **anonymousMethod2** will be applied against the implicitly generated class constructor.

```
Delegate Function ADelegate(ByVal value As Integer) As Boolean
Class AClass
    Dim anonymousMethod1 As ADelegate = Function(ByVal value As Integer) value > 5
    Shared anonymousMethod2 As ADelegate = Function(ByVal value As Integer) value > 5

    Sub Method1()
        anonymousMethod1(10)
        anonymousMethod2(10)
    End Sub
End Class
```

```
delegate void Delegate();

class Class
{
    Delegate anonymousMethod1 = delegate()
    {
        Console.WriteLine("");
    }

    static Delegate anonymousMethod2 = delegate()
    {
        Console.WriteLine("");
    }

    void Method()
    {
        anonymousMethod1();
        anonymousMethod2();
    }
}
```

A class could contain an inline anonymous method that assigns a value to a field that has multiple constructors. In this case, warnings and metrics are associated with all the constructors unless that constructor chains to another constructor in the same class.

For example, in the following class, any warnings that are found in the declaration of **anonymousMethod** should be raised against **Class(int)** and **Class(string)**, but not against **Class()**.

```
Delegate Function ADelegate(ByVal value As Integer) As Boolean

Class AClass

    Dim anonymousMethod As ADelegate = Function(ByVal value As Integer) value > 5

    SubNew()
        New(CStr(Nothing))
    End Sub

    Sub New(ByVal a As Integer)
    End Sub

    Sub New(ByVal a As String)
    End Sub
End Class
```

```
delegate void Delegate();

class Class
{
    Delegate anonymousMethod = delegate()
    {
        Console.WriteLine("");
    }

    Class() : this((string)null)
    {
    }

    Class(int a)
    {
    }

    Class(string a)
    {
    }
}
```

The warnings are raised against the constructors because the compiler outputs a unique method for every constructor that's not chained to another constructor. Because of this behavior, any violation that occurs in **anonymousMethod** must be suppressed separately. This also means that if a new constructor is introduced, warnings that were previously suppressed against **Class(int)** and **Class(string)** must also be suppressed against the new constructor.

You can work around this issue in one of two ways:

- Declare **anonymousMethod** in a common constructor that all constructors chain.
- Declare **anonymousMethod** in an initialization method that's called by all constructors.

## See also

- [Analyzing Managed Code Quality](#)

# How to: Create a Work Item for a Managed Code Defect

2/8/2019 • 2 minutes to read • [Edit Online](#)

You can use the work item tracking feature to log work item from within Visual Studio. To use this feature, your project must be part of an Azure DevOps project in Team Foundation.

## To create a work item for managed code defect

1. In the **Code Analysis** window, select the warning.
2. Choose **Actions**, then choose **Create Work Item** and choose the type of work item to create.

A new work item is created for you to specify the defect information.

## To create a work item for multiple managed code defects

1. In the **Error List**, select multiple warnings, and then right-click the warnings.
2. Point to **Create Work Item** and click the type of work item to create.

A single work item is created for all the selected warnings for you to specify the bug information.

# Use the C++ Core Guidelines checkers

3/21/2019 • 10 minutes to read • [Edit Online](#)

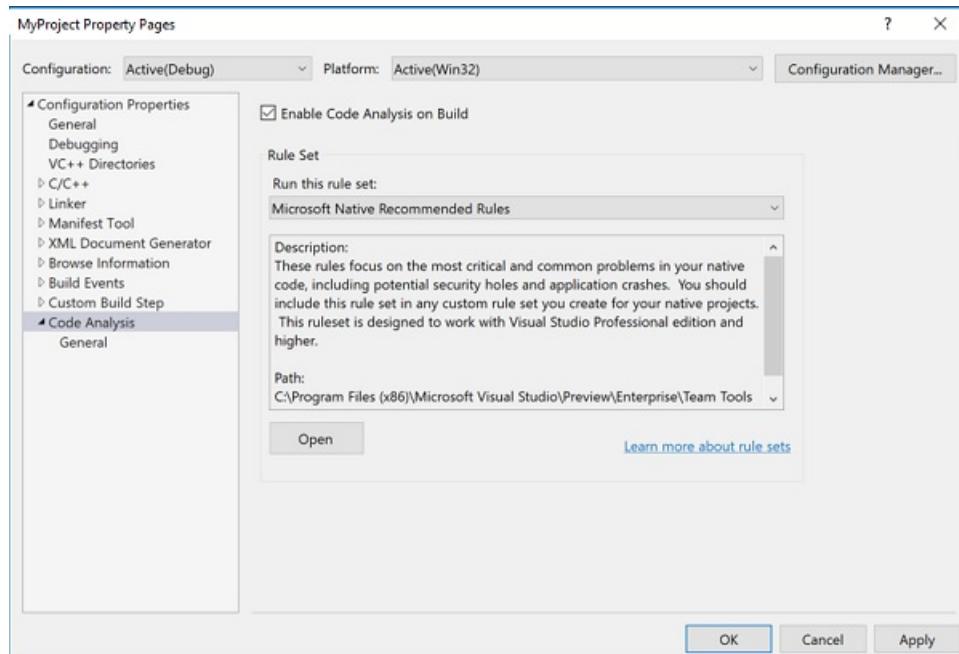
The C++ Core Guidelines are a portable set of guidelines, rules, and best practices about coding in C++ created by C++ experts and designers. Visual Studio currently supports a subset of these rules as part of its code analysis tools for C++. The core guideline checkers are installed by default in Visual Studio 2017 and Visual Studio 2019, and are available as a NuGet package for Visual Studio 2015.

## The C++ Core Guidelines Project

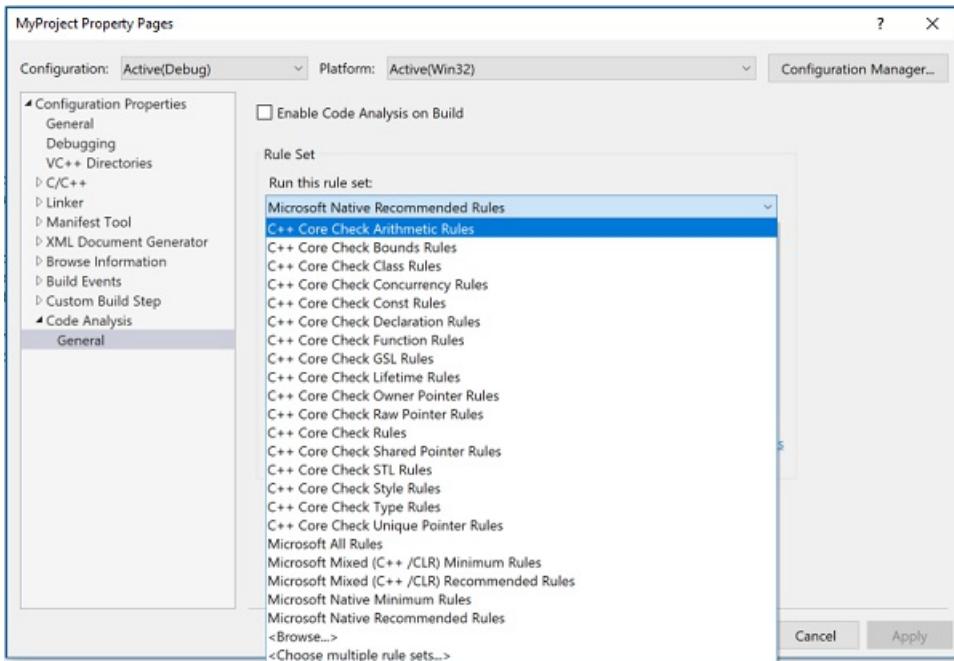
Created by Bjarne Stroustrup and others, the C++ Core Guidelines are a guide to using modern C++ safely and effectively. The Guidelines emphasize static type safety and resource safety. They identify ways to eliminate or minimize the most error-prone parts of the language, and suggest how to make your code simpler and more performant in a reliable way. These guidelines are maintained by the Standard C++ Foundation. To learn more, see the documentation, [C++ Core Guidelines](#), and access the C++ Core Guidelines documentation project files on [GitHub](#).

## Enable the C++ Core Check guidelines in Code Analysis

You can enable code analysis on your project by selecting the **Enable Code Analysis on Build** checkbox in the **Code Analysis** section of the **Property Pages** dialog for your project.



A subset of C++ Core Check rules is included in the Microsoft Native Recommended rule set that runs by default when code analysis is enabled. To enable additional Core Check rules, click on the dropdown and choose which rule sets you want to include:



## Examples

Here's an example of some of the issues that the C++ Core Check rules can find:

```
// CoreCheckExample.cpp
// Add CppCoreCheck package and enable code analysis in build for warnings.

int main()
{
    int arr[10];           // warning C26494
    int* p = arr;          // warning C26485

    [[gsl::suppress(bounds.1)]] // This attribute suppresses Bounds rule #1
    {
        int* q = p + 1;     // warning C26481 (suppressed)
        p = q++;            // warning C26481 (suppressed)
    }

    return 0;
}
```

This example demonstrates a few of the warnings that the C++ Core Check rules can find:

- C26494 is rule Type.5: Always initialize an object.
- C26485 is rule Bounds.3: No array-to-pointer decay.
- C26481 is rule Bounds.1: Don't use pointer arithmetic. Use `span` instead.

If the C++ Core Check code analysis rulesets are installed and enabled when you compile this code, the first two warnings are output, but the third is suppressed. Here's the build output from the example code:

```

1>----- Build started: Project: CoreCheckExample, Configuration: Debug Win32 -----
1> CoreCheckExample.cpp
1> CoreCheckExample.vcxproj -> C:\Users\username\documents\visual studio
2015\Projects\CoreCheckExample\Debug\CoreCheckExample.exe
1> CoreCheckExample.vcxproj -> C:\Users\username\documents\visual studio
2015\Projects\CoreCheckExample\Debug\CoreCheckExample.pdb (Full PDB)
c:\users\username\documents\visual studio
2015\projects\corecheckexample\corecheckexample\corecheckexample.cpp(6): warning C26494: Variable 'arr' is
uninitialized. Always initialize an object. (type.5: http://go.microsoft.com/fwlink/?LinkID=620421)
c:\users\username\documents\visual studio
2015\projects\corecheckexample\corecheckexample\corecheckexample.cpp(7): warning C26485: Expression 'arr': No
array to pointer decay. (bounds.3: http://go.microsoft.com/fwlink/?LinkID=620415)
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====

```

The C++ Core Guidelines are there to help you write better and safer code. However, if you have an instance where a rule or a profile shouldn't be applied, it's easy to suppress it directly in the code. You can use the `gsl::suppress` attribute to keep C++ Core Check from detecting and reporting any violation of a rule in the following code block. You can mark individual statements to suppress specific rules. You can even suppress the entire bounds profile by writing `[[gsl::suppress(bounds)]]` without including a specific rule number.

## Supported rule sets

As new rules are added to the C++ Core Guidelines Checker, the number of warnings that are produced for pre-existing code may increase. You can use predefined rule sets to filter which kinds of rules to enable. Reference topics for most rules are under [Visual Studio C++ Core Check Reference](#).

As of Visual Studio 2017 version 15.3, the supported rule sets are:

- **Owner Pointer Rules** enforce [resource-management checks related to owner<T>](#) from the C++ Core Guidelines.
- **Const Rules** enforce [const-related checks](#) from the C++ Core Guidelines.
- **Raw Pointer Rules** enforce [resource-management checks related to raw pointers](#) from the C++ Core Guidelines.
- **Unique Pointer Rules** enforce [resource-management checks related to types with unique pointer semantics](#) from the C++ Core Guidelines.
- **Bounds Rules** enforce the [Bounds profile](#) of the C++ Core Guidelines.
- **Type Rules** enforce the [Type profile](#) of the C++ Core Guidelines.

### Visual Studio 2017 version 15.5:

- **Class rules** A few rules that focus on proper use of special member functions and virtual specifications. This is a subset of checks recommended for [classes and class hierarchies](#).
- **Concurrency Rules** A single rule, which catches badly-declared guard objects. For more information, see [guidelines related to concurrency](#).
- **Declaration Rules** A couple of rules from the [interfaces guidelines](#) which focus on how global variables are declared.
- **Function Rules** Two checks that help with adoption of the `noexcept` specifier. This is a part of the guidelines for [clear function design and implementation](#).
- **Shared pointer Rules** As a part of [resource management](#) guidelines enforcement, we added a few rules specific to how shared pointers are passed into functions or used locally.
- **Style Rules** One simple but important check, which bans use of `goto`. This is the first step in improving of coding style and use of expressions and statements in C++.

## Visual Studio 2017 version 15.6:

- **Arithmetic Rules** Rules to detect arithmetic [overflow](#), [signed-unsigned operations](#) and [bit manipulation](#).

You can choose to limit warnings to just one or a few of the groups. The **Native Minimum** and **Native Recommended** rule sets include C++ Core Check rules in addition to other PREfast checks. To see the available rule sets, open the Project Properties dialog, select **Code Analysis\General**, open the dropdown in the **Rule Sets** combo-box, and pick **Choose multiple rule sets**. For more information about using Rule Sets in Visual Studio, see [Using Rule Sets to Group Code Analysis Rules](#).

## Macros

The C++ Core Guidelines Checker comes with a header file, which defines macros that make it easier to suppress entire categories of warnings in code:

```
ALL_CPPCORECHECK_WARNINGS
CPPCORECHECK_TYPE_WARNINGS
CPPCORECHECK_RAW_POINTER_WARNINGS
CPPCORECHECK_CONST_WARNINGS
CPPCORECHECK_OWNER_POINTER_WARNINGS
CPPCORECHECK_UNIQUE_POINTER_WARNINGS
CPPCORECHECK_BOUNDS_WARNINGS
```

These macros correspond to the rule sets and expand into a space-separated list of warning numbers. By using the appropriate pragma constructs, you can configure the effective set of rules that is interesting for a project or a section of code. In the following example, code analysis warns only about missing constant modifiers:

```
#include <CppCoreCheck\Warnings.h>
#pragma warning(disable: ALL_CPPCORECHECK_WARNINGS)
#pragma warning(default: CPPCORECHECK_CONST_WARNINGS)
```

## Attributes

The Microsoft Visual C++ compiler has a limited support for the GSL suppress attribute. It can be used to suppress warnings on expression and block statements inside of a function.

```
// Suppress only warnings from the 'r.11' rule in expression.
[[gsl::suppress(r.11)]] new int;

// Suppress all warnings from the 'r' rule group (resource management) in block.
[[gsl::suppress(r)]]
{
    new int;
}

// Suppress only one specific warning number.
// For declarations, you may need to use the surrounding block.
// Macros are not expanded inside of attributes.
// Use plain numbers instead of macros from the warnings.h.
[[gsl::suppress(26400)]]
{
    int *p = new int;
}
```

## Suppress analysis by using command-line options

Instead of #pragmas, you can use command-line options in the file's property page to suppress warnings for a

project or a single file. For example, to disable the warning 26400 for a file:

1. Right-click the file in **Solution Explorer**
2. Choose **Properties|C/C++|Command Line**
3. In the **Additional Options** window, add `/wd26400`.

You can use the command-line option to temporarily disable all code analysis for a file by specifying `/analyze-`. This produces warning *D9025 overriding '/analyze' with '/analyze-'*, which reminds you to re-enable code analysis later.

## Enable the C++ Core Guidelines Checker on specific project files

Sometimes it may be useful to do focused code analysis and still use the Visual Studio IDE. The following sample scenario can be used for large projects to save build time and to make it easier to filter results:

1. In the command shell set the `esp.extension` and `esp.annotationbuildlevel` environment variables.
2. To inherit these variables, open Visual Studio from the command shell.
3. Load your project and open its properties.
4. Enable code analysis, pick the appropriate rule sets, but do not enable code analysis extensions.
5. Go to the file you want to analyze with the C++ Core Guidelines Checker and open its properties.
6. Choose **C/C++\Command Line Options** and add `/analyze:plugin EspXEngine.dll`
7. Disable the use of precompiled header (**C/C++\Precompiled Headers**). This is necessary because the extensions engine may attempt to read its internal information from the precompiled header (PCH); if the PCH compiled with default project options, it will not be compatible.
8. Rebuild the project. The common PREFast checks should run on all files. Because the C++ Core Guidelines Checker is not enabled by default, it should only run on the file that is configured to use it.

## How to use the C++ Core Guidelines Checker outside of Visual Studio

You can use the C++ Core Guidelines checks in automated builds.

### MSBuild

The Native Code Analysis checker (PREFast) is integrated into MSBuild environment by custom targets files. You can use project properties to enable it, and add the C++ Core Guidelines Checker (which is based on PREFast):

```
<PropertyGroup>
    <EnableCppCoreCheck>true</EnableCppCoreCheck>
    <CodeAnalysisRuleSet>CppCoreCheckRules.ruleset</CodeAnalysisRuleSet>-->
    <RunCodeAnalysis>true</RunCodeAnalysis>
</PropertyGroup>
```

Make sure you add these properties before the import of the Microsoft.Cpp.targets file. You can pick specific rule sets or create a custom rule set or use the default rule set that includes other PREFast checks.

You can run the C++ Core Checker only on specified files by using the same approach as [described earlier](#), but using MSBuild files. The environment variables can be set by using the `BuildMacro` item:

```

<ItemGroup>
  <BuildMacro Include="Esp_AnnotationBuildLevel">
    <EnvironmentVariable>true</EnvironmentVariable>
    <Value>Ignore</Value>
  </BuildMacro>
  <BuildMacro Include="Esp_Extensions">
    <EnvironmentVariable>true</EnvironmentVariable>
    <Value>CppCoreCheck.dll</Value>
  </BuildMacro>
</ItemGroup>

```

If you don't want to modify the project file, you can pass properties on the command line:

```

msbuild /p:EnableCppCoreCheck=true /p:RunCodeAnalysis=true /p:CodeAnalysisRuleSet=CppCoreCheckRules.ruleset
...

```

## Non-MSBuild projects

If you use a build system that doesn't rely on MSBuild you can still run the checker, but you need to get familiar with some internals of the Code Analysis engine configuration. These internals are not guaranteed to be supported in the future.

You have to set a few environment variables and use proper command-line options for the compiler. It is better to work under the "Native Tools Command Prompt" environment so that you don't have to search for specific paths for the compiler, include directories, etc.

### 1. Environment variables

- `set esp.extensions=cppcorecheck.dll` This tells the engine to load the C++ Core Guidelines module.
- `set esp.annotationbuildlevel=ignore` This disables the logic that processes SAL annotations. Annotations don't affect code analysis in the C++ Core Guidelines Checker, yet their processing takes time (sometimes a long time). This setting is optional, but highly recommended.
- `set caexcludepath=%include%` We highly recommend that you disable warnings which fire on standard headers. You can add more paths here, for example the path to the common headers in your project.

### 2. Command line options

- `/analyze` Enables code analysis (consider also using `/analyze:only` and `/analyze:quiet`).
- `/analyze:plugin EspXEngine.dll` This option loads the Code Analysis Extensions engine into the PREfast. This engine, in turn, loads the C++ Core Guidelines Checker.

## Use the Guideline Support Library

The Guideline Support Library is designed to help you follow the Core Guidelines. The GSL includes definitions that let you replace error-prone constructs with safer alternatives. For example, you can replace a `T*, length` pair of parameters with the `span<T>` type. The GSL is available at <http://www.nuget.org/packages/Microsoft.Gsl>. The library is open-source, so you can view the sources, make comments, or contribute. The project can be found at <https://github.com/Microsoft/GSL>.

## Use the C++ Core Check guidelines in Visual Studio 2015 projects

If you use Visual Studio 2015, the C++ Core Check code analysis rule sets are not installed by default. You must perform some additional steps before you can enable the C++ Core Check code analysis tools in Visual Studio 2015. Microsoft provides support for Visual Studio 2015 projects by using a Nuget package. The package is named Microsoft.CppCoreCheck, and it is available at <http://www.nuget.org/packages/Microsoft.CppCoreCheck>. This package requires you have at least Visual Studio 2015 with Update 1 installed.

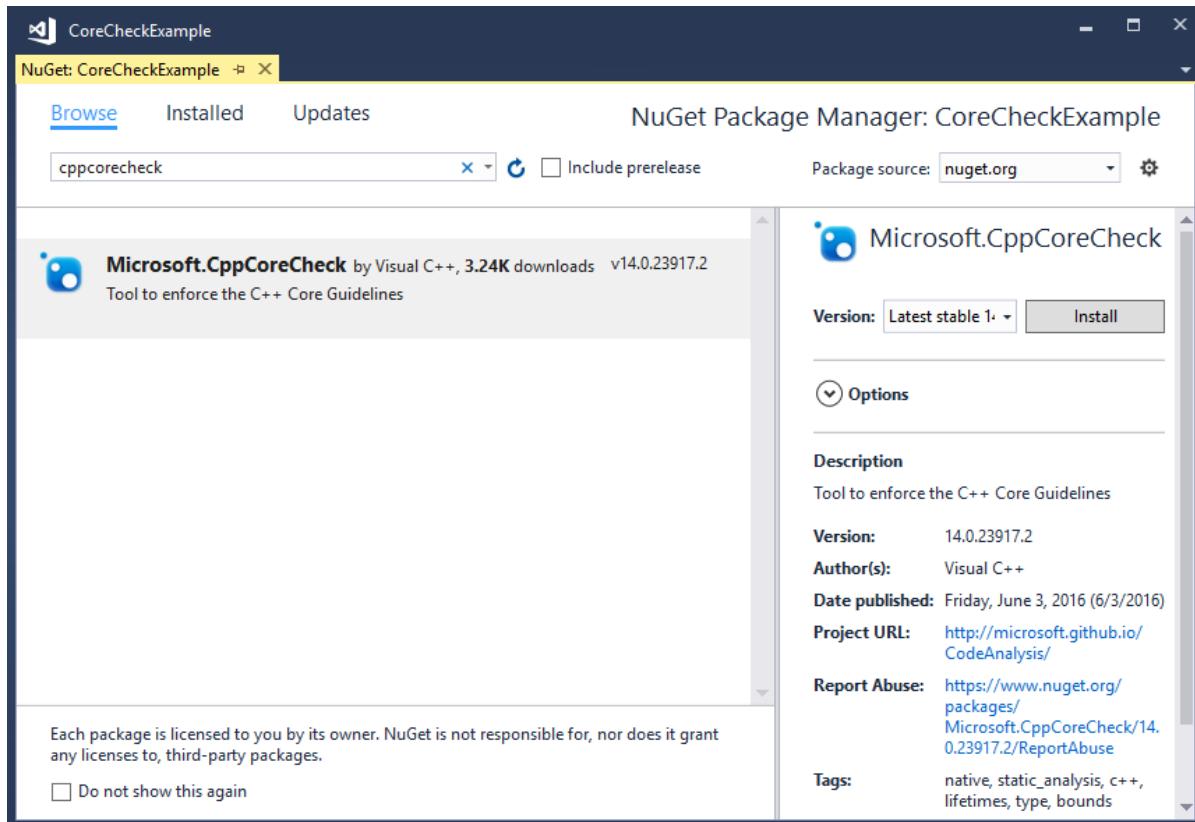
The package also installs another package as a dependency, a header-only Guideline Support Library (GSL). The

GSL is also available on GitHub at <https://github.com/Microsoft/GSL>.

Because of the way the code analysis rules are loaded, you must install the Microsoft.CppCoreCheck NuGet package into each C++ project that you want to check within Visual Studio 2015.

### To add the Microsoft.CppCoreCheck package to your project in Visual Studio 2015

1. In **Solution Explorer**, right-click to open the context menu of your project in the solution that you want to add the package to. Choose **Manage NuGet Packages** to open the **NuGet Package Manager**.
2. In the **NuGet Package Manager** window, search for Microsoft.CppCoreCheck.



3. Select the Microsoft.CppCoreCheck package and then choose the **Install** button to add the rules to your project.

The NuGet package adds an additional MSBuild *.targets* file to your project that is invoked when you enable code analysis on your project. This *.targets* file adds the C++ Core Check rules as an additional extension to the Visual Studio code analysis tool. When the package is installed, you can use the Property Pages dialog to enable or disable the released and experimental rules.

## See Also

- [Visual Studio C++ Core Check Reference](#)

# How to: Set Code Analysis Properties for C/C++ Projects

2/8/2019 • 2 minutes to read • [Edit Online](#)

You can configure which rules the code analysis tool uses to analyze the code in each configuration of your project. In addition, you can direct code analysis to suppress warnings from code that was generated and added to your project by a third-party tool.

## Code Analysis Property Page

The **Code Analysis** property page contains all code analysis configuration settings for a project. To open the code analysis property page for a project in **Solution Explorer**, right-click the project and then click **Properties**. Next, expand **Configuration Properties** and select the **Code Analysis** tab.

## Project Configuration and Platform

The **Configuration** list and **Platform** list lets you apply different code analysis settings to different project configuration and platform combinations. For example, you can direct code analysis to apply one set of rules to your project for debug builds and a different set for release builds.

## Enabling Code Analysis

You can decide whether to enable code analysis for your project by selecting **Enable Code Analysis For C/C++ on Build**. In combination with the **Configuration** list, you could, for example, decide to disable Code Analysis for debug builds and enable it for release builds.

If your project contains managed code, you can decide whether to enable or disable Code Analysis by selecting **Enable Code Analysis on Build**.

Code analysis is designed to help you improve the quality of your code and avoid common pitfalls. Therefore, consider carefully whether to disable code analysis. It is usually better to disable rule sets or individual rules that you do not want applied to your project.

## Generated Code

Developers frequently use tools to help develop applications quickly. These tools can generate code that is added to the project. You might want to see the rule violations that code analysis discovers in generated code. However, you might not want to see them if you do not want to maintain the code.

The **Suppress Results From Generated Code** check box on the **General** properties page lets you select whether you want to see code analysis warnings from managed code that is generated by a third-party tool.

## Rule Sets

If your project contains managed code, you can select the rules to apply in a code analysis by selecting a rule set from the **Run this rule set** list.

## See Also

- [Analyzing Managed Code Quality](#)

- Code Analysis for C/C++ Warnings

# Using SAL Annotations to Reduce C/C++ Code Defects

2/8/2019 • 2 minutes to read • [Edit Online](#)

SAL is the Microsoft source code annotation language. By using source code annotations, you can make the intent behind your code explicit. These annotations also enable automated static analysis tools to analyze your code more accurately, with significantly fewer false positives and false negatives.

The articles in this section of the documentation discuss aspects of SAL, provide reference for SAL syntax, and give examples of its use.

- [Understanding SAL](#)

Provides information and examples that show the core SAL annotations.

- [Annotating Function Parameters and Return Values](#)

Lists the SAL annotations for functions and function parameters.

- [Annotating Function Behavior](#)

Lists the SAL annotations for functions and function behavior.

- [Annotating Structs and Classes](#)

Lists the SAL annotations for structures and classes.

- [Annotating Locking Behavior](#)

Explains how to use SAL annotations with lock mechanisms.

- [Specifying When and Where an Annotation Applies](#)

Lists the SAL annotations that specify the condition or scope (placement) of other SAL annotations.

- [Intrinsic Functions](#)

Lists the intrinsic SAL annotations.

- [Best Practices and Examples](#)

Provides examples that show how to use SAL annotations. Also explains common pitfalls.

## Related Resources

[Code Analysis Team Blog](#)

## See Also

[SAL 2.0 Annotations for Windows Drivers](#)

# Understanding SAL

2/8/2019 • 11 minutes to read • [Edit Online](#)

The Microsoft source-code annotation language (SAL) provides a set of annotations that you can use to describe how a function uses its parameters, the assumptions that it makes about them, and the guarantees that it makes when it finishes. The annotations are defined in the header file `<sal.h>`. Visual Studio code analysis for C++ uses SAL annotations to modify its analysis of functions. For more information about SAL 2.0 for Windows driver development, see [SAL 2.0 Annotations for Windows Drivers](#).

Natively, C and C++ provide only limited ways for developers to consistently express intent and invariance. By using SAL annotations, you can describe your functions in greater detail so that developers who are consuming them can better understand how to use them.

## What Is SAL and Why Should You Use It?

Simply stated, SAL is an inexpensive way to let the compiler check your code for you.

### SAL Makes Code More Valuable

SAL can help you make your code design more understandable, both for humans and for code analysis tools. Consider this example that shows the C runtime function `memcpy`:

```
void * memcpy(
    void *dest,
    const void *src,
    size_t count
);
```

Can you tell what this function does? When a function is implemented or called, certain properties must be maintained to ensure program correctness. Just by looking at a declaration such as the one in the example, you don't know what they are. Without SAL annotations, you'd have to rely on documentation or code comments. Here's what the MSDN documentation for `memcpy` says:

"Copies count bytes of src to dest. If the source and destination overlap, the behavior of `memcpy` is undefined. Use `memmove` to handle overlapping regions. **Security Note:** Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see [Avoiding Buffer Overruns](#)."

The documentation contains a couple of bits of information that suggest that your code has to maintain certain properties to ensure program correctness:

- `memcpy` copies the `count` of bytes from the source buffer to the destination buffer.
- The destination buffer must be at least as large as the source buffer.

However, the compiler can't read the documentation or informal comments. It doesn't know that there is a relationship between the two buffers and `count`, and it also can't effectively guess about a relationship. SAL could provide more clarity about the properties and implementation of the function, as shown here:

```

void * memcpy(
    _Out_writes_bytes_all_(count) void *dest,
    _In_reads_bytes_(count) const void *src,
    size_t count
);

```

Notice that these annotations resemble the information in the MSDN documentation, but they are more concise and they follow a semantic pattern. When you read this code, you can quickly understand the properties of this function and how to avoid buffer overrun security issues. Even better, the semantic patterns that SAL provides can improve the efficiency and effectiveness of automated code analysis tools in the early discovery of potential bugs. Imagine that someone writes this buggy implementation of `wmemcpy`:

```

wchar_t * wmemcpy(
    _Out_writes_all_(count) wchar_t *dest,
    _In_reads_(count) const wchar_t *src,
    size_t count)
{
    size_t i;
    for (i = 0; i <= count; i++) { // BUG: off-by-one error
        dest[i] = src[i];
    }
    return dest;
}

```

This implementation contains a common off-by-one error. Fortunately, the code author included the SAL buffer size annotation—a code analysis tool could catch the bug by analyzing this function alone.

## SAL Basics

SAL defines four basic kinds of parameters, which are categorized by usage pattern.

CATEGORY	PARAMETER ANNOTATION	DESCRIPTION
<b>Input to called function</b>	<code>_In_</code>	Data is passed to the called function, and is treated as read-only.
<b>Input to called function, and output to caller</b>	<code>_Inout_</code>	Usable data is passed into the function and potentially is modified.
<b>Output to caller</b>	<code>_Out_</code>	The caller only provides space for the called function to write to. The called function writes data into that space.
<b>Output of pointer to caller</b>	<code>_Outptr_</code>	Like <b>Output to caller</b> . The value that's returned by the called function is a pointer.

These four basic annotations can be made more explicit in various ways. By default, annotated pointer parameters are assumed to be required—they must be non-NUL for the function to succeed. The most commonly used variation of the basic annotations indicates that a pointer parameter is optional—if it's NUL, the function can still succeed in doing its work.

This table shows how to distinguish between required and optional parameters:

	PARAMETERS ARE REQUIRED	PARAMETERS ARE OPTIONAL
<b>Input to called function</b>	<code>_In_</code>	<code>_In_opt_</code>
<b>Input to called function, and output to caller</b>	<code>_Inout_</code>	<code>_Inout_opt_</code>
<b>Output to caller</b>	<code>_Out_</code>	<code>_Out_opt_</code>
<b>Output of pointer to caller</b>	<code>_Outptr_</code>	<code>_Outptr_opt_</code>

These annotations help identify possible uninitialized values and invalid null pointer uses in a formal and accurate manner. Passing NULL to a required parameter might cause a crash, or it might cause a "failed" error code to be returned. Either way, the function cannot succeed in doing its job.

## SAL Examples

This section shows code examples for the basic SAL annotations.

### Using the Visual Studio Code Analysis Tool to Find Defects

In the examples, the Visual Studio Code Analysis tool is used together with SAL annotations to find code defects. Here's how to do that.

#### To use Visual Studio code analysis tools and SAL

1. In Visual Studio, open a C++ project that contains SAL annotations.
2. On the menu bar, choose **Build, Run Code Analysis on Solution**.

Consider the `_In_` example in this section. If you run code analysis on it, this warning is displayed:

**C6387 Invalid Parameter Value** 'pInt' could be '0': this does not adhere to the specification for the function 'InCallee'.

### Example: The `_In_` Annotation

The `_In_` annotation indicates that:

- The parameter must be valid and will not be modified.
- The function will only read from the single-element buffer.
- The caller must provide the buffer and initialize it.
- `_In_` specifies "read-only". A common mistake is to apply `_In_` to a parameter that should have the `_Inout_` annotation instead.
- `_In_` is allowed but ignored by the analyzer on non-pointer scalars.

```

void InCallee(_In_ int *pInt)
{
    int i = *pInt;
}

void GoodInCaller()
{
    int *pInt = new int;
    *pInt = 5;

    InCallee(pInt);
    delete pInt;
}

void BadInCaller()
{
    int *pInt = NULL;
    InCallee(pInt); // pInt should not be NULL
}

```

If you use Visual Studio Code Analysis on this example, it validates that the callers pass a non-Null pointer to an initialized buffer for `pInt`. In this case, `pInt` pointer cannot be NULL.

### **Example: The `_In_opt_` Annotation**

`_In_opt_` is the same as `_In_`, except that the input parameter is allowed to be NULL and, therefore, the function should check for this.

```

void GoodInOptCallee(_In_opt_ int *pInt)
{
    if(pInt != NULL) {
        int i = *pInt;
    }
}

void BadInOptCallee(_In_opt_ int *pInt)
{
    int i = *pInt; // Dereferencing NULL pointer 'pInt'
}

void InOptCaller()
{
    int *pInt = NULL;
    GoodInOptCallee(pInt);
    BadInOptCallee(pInt);
}

```

Visual Studio Code Analysis validates that the function checks for NULL before it accesses the buffer.

### **Example: The `_Out_` Annotation**

`_Out_` supports a common scenario in which a non-NULL pointer that points to an element buffer is passed in and the function initializes the element. The caller doesn't have to initialize the buffer before the call; the called function promises to initialize it before it returns.

```

void GoodOutCallee(_Out_ int *pInt)
{
    *pInt = 5;
}

void BadOutCallee(_Out_ int *pInt)
{
    // Did not initialize pInt buffer before returning!
}

void OutCaller()
{
    int *pInt = new int;
    GoodOutCallee(pInt);
    BadOutCallee(pInt);
    delete pInt;
}

```

Visual Studio Code Analysis Tool validates that the caller passes a non-NUL pointer to a buffer for `pInt` and that the buffer is initialized by the function before it returns.

### Example: The `_Out_opt_` Annotation

`_Out_opt_` is the same as `_out_`, except that the parameter is allowed to be NULL and, therefore, the function should check for this.

```

void GoodOutOptCallee(_Out_opt_ int *pInt)
{
    if (pInt != NULL) {
        *pInt = 5;
    }
}

void BadOutOptCallee(_Out_opt_ int *pInt)
{
    *pInt = 5; // Dereferencing NULL pointer 'pInt'
}

void OutOptCaller()
{
    int *pInt = NULL;
    GoodOutOptCallee(pInt);
    BadOutOptCallee(pInt);
}

```

Visual Studio Code Analysis validates that this function checks for NULL before `pInt` is dereferenced, and if `pInt` is not NULL, that the buffer is initialized by the function before it returns.

### Example: The `_Inout_` Annotation

`_Inout_` is used to annotate a pointer parameter that may be changed by the function. The pointer must point to valid initialized data before the call, and even if it changes, it must still have a valid value on return. The annotation specifies that the function may freely read from and write to the one-element buffer. The caller must provide the buffer and initialize it.

#### NOTE

Like `_Out_`, `_Inout_` must apply to a modifiable value.

```

void InOutCallee(_Inout_ int *pInt)
{
    int i = *pInt;
    *pInt = 6;
}

void InOutCaller()
{
    int *pInt = new int;
    *pInt = 5;
    InOutCallee(pInt);
    delete pInt;
}

void BadInOutCaller()
{
    int *pInt = NULL;
    InOutCallee(pInt); // 'pInt' should not be NULL
}

```

Visual Studio Code Analysis validates that callers pass a non-NULL pointer to an initialized buffer for `pInt`, and that, before return, `pInt` is still non-NULL and the buffer is initialized.

#### **Example: The `_Inout_opt_` Annotation**

`_Inout_opt_` is the same as `_Inout_`, except that the input parameter is allowed to be NULL and, therefore, the function should check for this.

```

void GoodInOutOptCallee(_Inout_opt_ int *pInt)
{
    if(pInt != NULL) {
        int i = *pInt;
        *pInt = 6;
    }
}

void BadInOutOptCallee(_Inout_opt_ int *pInt)
{
    int i = *pInt; // Dereferencing NULL pointer 'pInt'
    *pInt = 6;
}

void InOutOptCaller()
{
    int *pInt = NULL;
    GoodInOutOptCallee(pInt);
    BadInOutOptCallee(pInt);
}

```

Visual Studio Code Analysis validates that this function checks for NULL before it accesses the buffer, and if `pInt` is not NULL, that the buffer is initialized by the function before it returns.

#### **Example: The `_Outptr_` Annotation**

`_Outptr_` is used to annotate a parameter that's intended to return a pointer. The parameter itself should not be NULL, and the called function returns a non-NULL pointer in it and that pointer points to initialized data.

```

void GoodOutPtrCallee(_Outptr_ int **pInt)
{
    int *pInt2 = new int;
    *pInt2 = 5;

    *pInt = pInt2;
}

void BadOutPtrCallee(_Outptr_ int **pInt)
{
    int *pInt2 = new int;
    // Did not initialize pInt buffer before returning!
    *pInt = pInt2;
}

void OutPtrCaller()
{
    int *pInt = NULL;
    GoodOutPtrCallee(&pInt);
    BadOutPtrCallee(&pInt);
}

```

Visual Studio Code Analysis validates that the caller passes a non-NULL pointer for `*pInt`, and that the buffer is initialized by the function before it returns.

### **Example: The `_Outptr_opt_` Annotation**

`_Outptr_opt_` is the same as `_Outptr_`, except that the parameter is optional—the caller can pass in a NULL pointer for the parameter.

```

void GoodOutPtrOptCallee(_Outptr_opt_ int **pInt)
{
    int *pInt2 = new int;
    *pInt2 = 6;

    if(pInt != NULL) {
        *pInt = pInt2;
    }
}

void BadOutPtrOptCallee(_Outptr_opt_ int **pInt)
{
    int *pInt2 = new int;
    *pInt2 = 6;
    *pInt = pInt2; // Dereferencing NULL pointer 'pInt'
}

void OutPtrOptCaller()
{
    int **ppInt = NULL;
    GoodOutPtrOptCallee(ppInt);
    BadOutPtrOptCallee(ppInt);
}

```

Visual Studio Code Analysis validates that this function checks for NULL before `*pInt` is dereferenced, and that the buffer is initialized by the function before it returns.

### **Example: The `_Success_` Annotation in Combination with `_Out_`**

Annotations can be applied to most objects. In particular, you can annotate a whole function. One of the most obvious characteristics of a function is that it can succeed or fail. But like the association between a buffer and its size, C/C++ cannot express function success or failure. By using the `_Success_` annotation, you can say what success for a function looks like. The parameter to the `_Success_` annotation is just an expression that when it is

true indicates that the function has succeeded. The expression can be anything that the annotation parser can handle. The effects of the annotations after the function returns are only applicable when the function succeeds. This example shows how `_Success_` interacts with `_out_` to do the right thing. You can use the keyword `return` to represent the return value.

```
_Success_(return != false) // Can also be stated as _Success_(return)
bool GetValue(_Out_ int *pInt, bool flag)
{
    if(flag) {
        *pInt = 5;
        return true;
    } else {
        return false;
    }
}
```

The `_out_` annotation causes Visual Studio Code Analysis to validate that the caller passes a non-NUL pointer to a buffer for `pInt`, and that the buffer is initialized by the function before it returns.

## SAL Best Practice

### Adding Annotations to Existing Code

SAL is a powerful technology that can help you improve the security and reliability of your code. After you learn SAL, you can apply the new skill to your daily work. In new code, you can use SAL-based specifications by design throughout; in older code, you can add annotations incrementally and thereby increase the benefits every time you update.

Microsoft public headers are already annotated. Therefore, we suggest that in your projects you first annotate leaf node functions and functions that call Win32 APIs to get the most benefit.

### When Do I Annotate?

Here are some guidelines:

- Annotate all pointer parameters.
- Annotate value-range annotations so that Code Analysis can ensure buffer and pointer safety.
- Annotate locking rules and locking side effects. For more information, see [Annotating Locking Behavior](#).
- Annotate driver properties and other domain-specific properties.

Or you can annotate all parameters to make your intent clear throughout and to make it easy to check that annotations have been done.

## Related Resources

[Code Analysis Team Blog](#)

## See Also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Specifying When and Where an Annotation Applies](#)

- Best Practices and Examples

# Annotating Function Parameters and Return Values

2/8/2019 • 12 minutes to read • [Edit Online](#)

This article describes typical uses of annotations for simple function parameters—scalars, and pointers to structures and classes—and most kinds of buffers. This article also shows common usage patterns for annotations. For additional annotations that are related to functions, see [Annotating Function Behavior](#)

## Pointer Parameters

For the annotations in the following table, when a pointer parameter is being annotated, the analyzer reports an error if the pointer is null. This applies to pointers and to any data item that's pointed to.

### Annotations and Descriptions

- `_In_`

Annotates input parameters that are scalars, structures, pointers to structures and the like. Explicitly may be used on simple scalars. The parameter must be valid in pre-state and will not be modified.

- `_Out_`

Annotates output parameters that are scalars, structures, pointers to structures and the like. Do not apply this to an object that cannot return a value—for example, a scalar that's passed by value. The parameter does not have to be valid in pre-state but must be valid in post-state.

- `_Inout_`

Annotates a parameter that will be changed by the function. It must be valid in both pre-state and post-state, but is assumed to have different values before and after the call. Must apply to a modifiable value.

- `_In_z_`

A pointer to a null-terminated string that's used as input. The string must be valid in pre-state. Variants of `PSTR`, which already have the correct annotations, are preferred.

- `_Inout_z_`

A pointer to a null-terminated character array that will be modified. It must be valid before and after the call, but the value is assumed to have changed. The null terminator may be moved, but only the elements up to the original null terminator may be accessed.

- `_In_reads_(s)`

`_In_reads_bytes_(s)`

A pointer to an array, which is read by the function. The array is of size `s` elements, all of which must be valid.

The `_bytes_` variant gives the size in bytes instead of elements. Use this only when the size cannot be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_In_reads_z_(s)`

A pointer to an array that is null-terminated and has a known size. The elements up to the null terminator

—or `s` if there is no null terminator—must be valid in pre-state. If the size is known in bytes, scale `s` by the element size.

- `_In_reads_or_z_(s)`

A pointer to an array that is null-terminated or has a known size, or both. The elements up to the null terminator—or `s` if there is no null terminator—must be valid in pre-state. If the size is known in bytes, scale `s` by the element size. (Used for the `strn` family.)

- `_Out_writes_(s)`

`_Out_writes_bytes_(s)`

A pointer to an array of `s` elements (resp. bytes) that will be written by the function. The array elements do not have to be valid in pre-state, and the number of elements that are valid in post-state is unspecified. If there are annotations on the parameter type, they are applied in post-state. For example, consider the following code.

```
typedef _Null_terminated_ wchar_t *PWSTR; void MyStringCopy(_Out_writes_(size) PWSTR p1, _In_ size_t size, _In_ PWSTR p2);
```

In this example, the caller provides a buffer of `size` elements for `p1`. `MyStringCopy` makes some of those elements valid. More importantly, the `_Null_terminated_` annotation on `PWSTR` means that `p1` is null-terminated in post-state. In this way, the number of valid elements is still well-defined, but a specific element count is not required.

The `_bytes_` variant gives the size in bytes instead of elements. Use this only when the size cannot be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_Out_writes_z_(s)`

A pointer to an array of `s` elements. The elements do not have to be valid in pre-state. In post-state, the elements up through the null terminator—which must be present—must be valid. If the size is known in bytes, scale `s` by the element size.

- `_Inout_updates_(s)`

`_Inout_updates_bytes_(s)`

A pointer to an array, which is both read and written to in the function. It is of size `s` elements, and valid in pre-state and post-state.

The `_bytes_` variant gives the size in bytes instead of elements. Use this only when the size cannot be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_Inout_updates_z_(s)`

A pointer to an array that is null-terminated and has a known size. The elements up through the null terminator—which must be present—must be valid in both pre-state and post-state. The value in the post-state is presumed to be different from the value in the pre-state; this includes the location of the null terminator. If the size is known in bytes, scale `s` by the element size.

- `_Out_writes_to_(s,c)`

`_Out_writes_bytes_to_(s,c)`

`_Out_writes_all_(s)`

```
_Out_writes_bytes_all_(s)
```

A pointer to an array of `s` elements. The elements do not have to be valid in pre-state. In post-state, the elements up to the `c`-th element must be valid. If the size is known in bytes, scale `s` and `c` by the element size or use the `_bytes_` variant, which is defined as:

```
_Out_writes_to_(Old_(s), Old_(s)) _Out_writes_bytes_to_(Old_(s), Old_(s))
```

In other words, every element that exists in the buffer up to `s` in the pre-state is valid in the post-state. For example:

```
void *memcpy(_Out_writes_bytes_all_(s) char *p1, _In_reads_bytes_(s) char *p2, _In_ int s); void *  
wordcpy(_Out_writes_all_(s) DWORD *p1, _In_reads_(s) DWORD *p2, _In_ int s);
```

- `_Inout_updates_to_(s,c)`

```
_Inout_updates_bytes_to_(s,c)
```

A pointer to an array, which is both read and written by the function. It is of size `s` elements, all of which must be valid in pre-state, and `c` elements must be valid in post-state.

The `_bytes_` variant gives the size in bytes instead of elements. Use this only when the size cannot be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_Inout_updates_z_(s)`

A pointer to an array that is null-terminated and has a known size. The elements up through the null terminator—which must be present—must be valid in both pre-state and post-state. The value in the post-state is presumed to be different from the value in the pre-state; this includes the location of the null terminator. If the size is known in bytes, scale `s` by the element size.

- `_Out_writes_to_(s,c)`

```
_Out_writes_bytes_to_(s,c)
```

```
_Out_writes_all_(s)
```

```
_Out_writes_bytes_all_(s)
```

A pointer to an array of `s` elements. The elements do not have to be valid in pre-state. In post-state, the elements up to the `c`-th element must be valid. If the size is known in bytes, scale `s` and `c` by the element size or use the `_bytes_` variant, which is defined as:

```
_Out_writes_to_(Old_(s), Old_(s)) _Out_writes_bytes_to_(Old_(s), Old_(s))
```

In other words, every element that exists in the buffer up to `s` in the pre-state is valid in the post-state. For example:

```
void *memcpy(_Out_writes_bytes_all_(s) char *p1, _In_reads_bytes_(s) char *p2, _In_ int s); void *  
wordcpy(_Out_writes_all_(s) DWORD *p1, _In_reads_(s) DWORD *p2, _In_ int s);
```

- `_Inout_updates_to_(s,c)`

```
_Inout_updates_bytes_to_(s,c)
```

A pointer to an array, which is both read and written by the function. It is of size `s` elements, all of which must be valid in pre-state, and `c` elements must be valid in post-state.

The `_bytes_` variant gives the size in bytes instead of elements. Use this only when the size cannot be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function

that uses `wchar_t` would.

- `_Inout_updates_all_(s)`

`_Inout_updates_bytes_all_(s)`

A pointer to an array, which is both read and written by the function of size `s` elements. Defined as equivalent to:

`_Inout_updates_to_(old_(s), old_(s)) _Inout_updates_bytes_to_(old_(s), old_(s))`

In other words, every element that exists in the buffer up to `s` in the pre-state is valid in the pre-state and post-state.

The `_bytes_` variant gives the size in bytes instead of elements. Use this only when the size cannot be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

- `_In_reads_to_ptr_(p)`

A pointer to an array for which the expression `p - _curr_` (that is, `p` minus `_curr_`) is defined by the appropriate language standard. The elements prior to `p` must be valid in pre-state.

- `_In_reads_to_ptr_z_(p)`

A pointer to a null-terminated array for which the expression `p - _curr_` (that is, `p` minus `_curr_`) is defined by the appropriate language standard. The elements prior to `p` must be valid in pre-state.

- `_Out_writes_to_ptr_(p)`

A pointer to an array for which the expression `p - _curr_` (that is, `p` minus `_curr_`) is defined by the appropriate language standard. The elements prior to `p` do not have to be valid in pre-state and must be valid in post-state.

- `_Out_writes_to_ptr_z_(p)`

A pointer to a null-terminated array for which the expression `p - _curr_` (that is, `p` minus `_curr_`) is defined by the appropriate language standard. The elements prior to `p` do not have to be valid in pre-state and must be valid in post-state.

## Optional Pointer Parameters

When a pointer parameter annotation includes `_opt_`, it indicates that the parameter may be null. Otherwise, the annotation performs the same as the version that doesn't include `_opt_`. Here is a list of the `_opt_` variants of the pointer parameter annotations:

<code>_In_opt_</code>	<code>_Out_writes_opt_</code>	<code>_Inout_updates_to_opt_</code>
<code>_Out_opt_</code>	<code>_Out_writes_opt_z_</code>	<code>_Inout_updates_bytes_to_opt_</code>
<code>_Inout_opt_</code>	<code>_Inout_updates_opt_</code>	<code>_Inout_updates_all_opt_</code>
<code>_In_opt_z_</code>	<code>_Inout_updates_bytes_opt_</code>	<code>_Inout_updates_bytes_all_opt_</code>
<code>_Inout_opt_z_</code>	<code>_Inout_updates_opt_z_</code>	<code>_In_reads_to_ptr_opt_</code>
<code>_In_reads_opt_</code>	<code>_Out_writes_to_opt_</code>	<code>_In_reads_to_ptr_opt_z_</code>
<code>_In_reads_bytes_opt_</code>	<code>_Out_writes_bytes_to_opt_</code>	<code>_Out_writes_to_ptr_opt_</code>
<code>_In_reads_opt_z_</code>	<code>_Out_writes_all_opt_</code>	<code>_Out_writes_to_ptr_opt_z_</code>
	<code>_Out_writes_bytes_all_opt_</code>	

## Output Pointer Parameters

Output pointer parameters require special notation to disambiguate null-ness on the parameter and the pointed-to location.

### Annotations and Descriptions

- `_Outptr_`

Parameter cannot be null, and in the post-state the pointed-to location cannot be null and must be valid.

- `_Outptr_opt_`

Parameter may be null, but in the post-state the pointed-to location cannot be null and must be valid.

- `_Outptr_result_maybenull_`

Parameter cannot be null, and in the post-state the pointed-to location can be null.

- `_Outptr_opt_result_maybenull_`

Parameter may be null, and in the post-state the pointed-to location can be null.

In the following table, additional substrings are inserted into the annotation name to further qualify the meaning of the annotation. The various substrings are `_z`, `_COM_`, `_buffer_`, `_bytebuffer_`, and `_to_`.

### IMPORTANT

If the interface that you are annotating is COM, use the COM form of these annotations. Do not use the COM annotations with any other type interface.

### Annotations and Descriptions

- `_Outptr_result_z_`

`_Outptr_opt_result_z_`

`_Outptr_result_maybenull_z_`

`_Outptr_opt_result_maybenull_z_`

The returned pointer has the `_Null_terminated_` annotation.

- `_COM_Outptr_`  
`_COM_Outptr_opt_`  
`_COM_Outptr_result_maybenull_`  
`_COM_Outptr_opt_result_maybenull_`

The returned pointer has COM semantics, and therefore carries an `_On_failure_` post-condition that the returned pointer is null.

- `_Outptr_result_buffer_(s)`  
`_Outptr_result_bytebuffer_(s)`  
`_Outptr_opt_result_buffer_(s)`  
`_Outptr_opt_result_bytebuffer_(s)`

The returned pointer points to a valid buffer of size `s` elements or bytes.

- `_Outptr_result_buffer_to_(s, c)`  
`_Outptr_result_bytebuffer_to_(s, c)`  
`_Outptr_opt_result_buffer_to_(s,c)`  
`_Outptr_opt_result_bytebuffer_to_(s,c)`

The returned pointer points to a buffer of size `s` elements or bytes, of which the first `c` are valid.

Certain interface conventions presume that output parameters are nullified on failure. Except for explicitly COM code, the forms in the following table are preferred. For COM code, use the corresponding COM forms that are listed in the previous section.

## Annotations and Descriptions

- `_Result_nullonfailure_`  
Modifies other annotations. The result is set to null if the function fails.
- `_Result_zeroonfailure_`  
Modifies other annotations. The result is set to zero if the function fails.
- `_Outptr_result_nullonfailure_`  
The returned pointer points to a valid buffer if the function succeeds, or null if the function fails. This annotation is for a non-optional parameter.
- `_Outptr_opt_result_nullonfailure_`  
The returned pointer points to a valid buffer if the function succeeds, or null if the function fails. This annotation is for an optional parameter.
- `_Outref_result_nullonfailure_`  
The returned pointer points to a valid buffer if the function succeeds, or null if the function fails. This annotation is for a reference parameter.

# Output Reference Parameters

A common use of the reference parameter is for output parameters. For simple output reference parameters—for example, `int& — _out_` provides the correct semantics. However, when the output value is a pointer—for example `int *&`—the equivalent pointer annotations like `_Outptr_ int **` don't provide the correct semantics. To concisely express the semantics of output reference parameters for pointer types, use these composite annotations:

## Annotations and Descriptions

- `_Outref_`

Result must be valid in post-state and cannot be null.
- `_Outref_result_maybenull_`

Result must be valid in post-state, but may be null in post-state.
- `_Outref_result_buffer_(s)`

Result must be valid in post-state and cannot be null. Points to valid buffer of size `s` elements.
- `_Outref_result_bytebuffer_(s)`

Result must be valid in post-state and cannot be null. Points to valid buffer of size `s` bytes.
- `_Outref_result_buffer_to_(s, c)`

Result must be valid in post-state and cannot be null. Points to buffer of `s` elements, of which the first `c` are valid.
- `_Outref_result_bytebuffer_to_(s, c)`

Result must be valid in post-state and cannot be null. Points to buffer of `s` bytes of which the first `c` are valid.
- `_Outref_result_buffer_all_(s)`

Result must be valid in post-state and cannot be null. Points to valid buffer of size `s` valid elements.
- `_Outref_result_bytebuffer_all_(s)`

Result must be valid in post-state and cannot be null. Points to valid buffer of `s` bytes of valid elements.
- `_Outref_result_buffer_maybenull_(s)`

Result must be valid in post-state, but may be null in post-state. Points to valid buffer of size `s` elements.
- `_Outref_result_bytebuffer_maybenull_(s)`

Result must be valid in post-state, but may be null in post-state. Points to valid buffer of size `s` bytes.
- `_Outref_result_buffer_to_maybenull_(s, c)`

Result must be valid in post-state, but may be null in post-state. Points to buffer of `s` elements, of which the first `c` are valid.
- `_Outref_result_bytebuffer_to_maybenull_(s, c)`

Result must be valid in post-state, but may be null in post state. Points to buffer of `s` bytes of which the first `c` are valid.

- `_Outref_result_buffer_all_maybenull_(s)`

Result must be valid in post-state, but may be null in post state. Points to valid buffer of size `s` valid elements.

- `_Outref_result_bytbuffer_all_maybenull_(s)`

Result must be valid in post-state, but may be null in post state. Points to valid buffer of `s` bytes of valid elements.

## Return Values

The return value of a function resembles an `_out_` parameter but is at a different level of de-reference, and you don't have to consider the concept of the pointer to the result. For the following annotations, the return value is the annotated object—a scalar, a pointer to a struct, or a pointer to a buffer. These annotations have the same semantics as the corresponding `_out_` annotation.

<code>_Ret_z_</code>	<code>_Ret_maybenull_</code>
<code>_Ret_writes_(s)</code>	<code>_Ret_maybenull_z_</code>
<code>_Ret_writes_bytes_(s)</code>	<code>_Ret_null_</code>
<code>_Ret_writes_z_(s)</code>	<code>_Ret_notnull_</code>
<code>_Ret_writes_to_(s,c)</code>	<code>_Ret_writes_bytes_to_</code>
<code>_Ret_writes_maybenull_(s)</code>	<code>_Ret_writes_bytes_maybenull_</code>
<code>_Ret_writes_to_maybenull_(s)</code>	<code>_Ret_writes_bytes_to_maybenull_</code>
<code>_Ret_writes_maybenull_z_(s)</code>	

## Other Common Annotations

### Annotations and Descriptions

- `_In_range_(low, hi)`

`_Out_range_(low, hi)`

`_Ret_range_(low, hi)`

`_Deref_in_range_(low, hi)`

`_Deref_out_range_(low, hi)`

`_Deref inout_range_(low, hi)`

`_Field_range_(low, hi)`

The parameter, field, or result is in the range (inclusive) from `low` to `hi`. Equivalent to

`_Satisfies_(_Curr_ >= low && _Curr_ <= hi)` that is applied to the annotated object together with the appropriate pre-state or post-state conditions.

### IMPORTANT

Although the names contain "in" and "out", the semantics of `_In_` and `_Out_` do **not** apply to these annotations.

- `_Pre_equal_to_(expr)`

`_Post_equal_to_(expr)`

The annotated value is exactly `expr`. Equivalent to `_Satisfies_(_Curr_ == expr)` that is applied to the annotated object together with the appropriate pre-state or post-state conditions.

- `_Struct_size_bytes_(size)`

Applies to a struct or class declaration. Indicates that a valid object of that type may be larger than the declared type, with the number of bytes being given by `size`. For example:

```
typedef _Struct_size_bytes_(nSize) struct MyStruct { size_t nSize; ... };
```

The buffer size in bytes of a parameter `pM` of type `MyStruct *` is then taken to be:

```
min(pM->nSize, sizeof(MyStruct))
```

## Related Resources

[Code Analysis Team Blog](#)

## See Also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Specifying When and Where an Annotation Applies](#)
- [Intrinsic Functions](#)
- [Best Practices and Examples](#)

# Annotating Function Behavior

2/8/2019 • 3 minutes to read • [Edit Online](#)

In addition to annotating [function parameters and return values](#), you can annotate properties of the whole function.

## Function Annotations

The following annotations apply to the function as a whole and describe how it behaves or what it expects to be true.

ANNOTATION	DESCRIPTION
<code>_Called_from_function_class_(name)</code>	<p>Not intended to stand alone; instead, it is a predicate to be used with the <code>_when_</code> annotation. For more information, see <a href="#">Specifying When and Where an Annotation Applies</a>.</p> <p>The <code>name</code> parameter is an arbitrary string that also appears in a <code>_Function_class_</code> annotation in the declaration of some functions. <code>_Called_from_function_class_</code> returns nonzero if the function that is currently being analyzed is annotated by using <code>_Function_class_</code> that has the same value of <code>name</code>; otherwise, it returns zero.</p>
<code>_Check_return_</code>	Annotates a return value and states that the caller should inspect it. The checker reports an error if the function is called in a void context.
<code>_Function_class_(name)</code>	The <code>name</code> parameter is an arbitrary string that is designated by the user. It exists in a namespace that is distinct from other namespaces. A function, function pointer, or—most usefully—a function pointer type may be designated as belonging to one or more function classes.
<code>_Raises_SEH_exception_</code>	Annotates a function that always raises a structured exception handler (SEH) exception, subject to <code>_when_</code> and <code>_on_failure_</code> conditions. For more information, see <a href="#">Specifying When and Where an Annotation Applies</a> .
<code>_Maybe_raises_SEH_exception_</code>	Annotates a function that may optionally raise an SEH exception, subject to <code>_when_</code> and <code>_on_failure_</code> conditions.
<code>_Must_inspect_result_</code>	Annotates any output value, including the return value, parameters, and globals. The analyzer reports an error if the value in the annotated object is not subsequently inspected. "Inspection" includes whether it is used in a conditional expression, is assigned to an output parameter or global, or is passed as a parameter. For return values, <code>_Must_inspect_result_</code> implies <code>_Check_return_</code> .

ANNOTATION	DESCRIPTION
<code>_Use_decl_annotations_</code>	May be used on a function definition (also known as a function body) in place of the list of annotations in the header. When <code>_Use_decl_annotations_</code> is used, the annotations that appear on an in-scope header for the same function are used as if they are also present in the definition that has the <code>_Use_decl_annotations_</code> annotation.

## Success/Failure Annotations

A function can fail, and when it does, its results may be incomplete or differ from the results when the function succeeds. The annotations in the following list provide ways to express the failure behavior. To use these annotations, you must enable them to determine success; therefore, a `_Success_` annotation is required. Notice that `NTSTATUS` and `HRESULT` already have a `_Success_` annotation built into them; however, if you specify your own `_Success_` annotation on `NTSTATUS` or `HRESULT`, it overrides the built-in annotation.

ANNOTATION	DESCRIPTION
<code>_Always_(anno_list)</code>	Equivalent to <code>anno_list _On_failure_(anno_list)</code> ; that is, the annotations in <code>anno_list</code> apply whether or not the function succeeds.
<code>_On_failure_(anno_list)</code>	To be used only when <code>_Success_</code> is also used to annotate the function—either explicitly, or implicitly through <code>_Return_type_success_</code> on a typedef. When the <code>_On_failure_</code> annotation is present on a function parameter or return value, each annotation in <code>anno_list</code> ( <code>anno</code> ) behaves as if it were coded as <code>_when_(!expr, anno)</code> , where <code>expr</code> is the parameter to the required <code>_Success_</code> annotation. This means that the implied application of <code>_Success_</code> to all post-conditions does not apply for <code>_On_failure_</code> .
<code>_Return_type_success_(expr)</code>	May be applied to a typedef. Indicates that all functions that return that type and do not explicitly have <code>_Success_</code> are annotated as if they had <code>_Success_(expr)</code> . <code>_Return_type_success_</code> cannot be used on a function or a function pointer typedef.
<code>_Success_(expr)</code>	<code>expr</code> is an expression that yields an rvalue. When the <code>_Success_</code> annotation is present on a function declaration or definition, each annotation ( <code>anno</code> ) on the function and in post-condition behaves as if it were coded as <code>_When_(expr, anno)</code> . The <code>_Success_</code> annotation may be used only on a function, not on its parameters or return type. There can be at most one <code>_Success_</code> annotation on a function, and it cannot be in any <code>_When_</code> , <code>_At_</code> , or <code>_Group_</code> . For more information, see <a href="#">Specifying When and Where an Annotation Applies</a> .

## See also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)

- [Annotating Function Parameters and Return Values](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Specifying When and Where an Annotation Applies](#)
- [Intrinsic Functions](#)
- [Best Practices and Examples](#)

# Annotating Structs and Classes

2/8/2019 • 2 minutes to read • [Edit Online](#)

You can annotate struct and class members by using annotations that act like invariants—they are presumed to be true at any function call or function entry/exit that involves the enclosing structure as a parameter or a result value.

## Struct and Class Annotations

- `_Field_range_(low, high)`

The field is in the range (inclusive) from `low` to `high`. Equivalent to `_Satisfies_(_Curr_ >= low && _Curr_ <= high)` applied to the annotated object by using the appropriate pre or post conditions.

- `_Field_size_(size)`, `_Field_size_opt_(size)`, `_Field_size_bytes_(size)`, `_Field_size_bytes_opt_(size)`

A field that has a writable size in elements (or bytes) as specified by `size`.

- `_Field_size_part_(size, count)`, `_Field_size_part_opt_(size, count)`,  
`_Field_size_bytes_part_(size, count)`, `_Field_size_bytes_part_opt_(size, count)`

A field that has a writable size in elements (or bytes) as specified by `size`, and the `count` of those elements (bytes) that are readable.

- `_Field_size_full_(size)`, `_Field_size_full_opt_(size)`, `_Field_size_bytes_full_(size)`,  
`_Field_size_bytes_full_opt_(size)`

A field that has both readable and writable size in elements (or bytes) as specified by `size`.

- `_Field_z_`

A field that has a null-terminated string.

- `_Struct_size_bytes_(size)`

Applies to struct or class declaration. Indicates that a valid object of that type may be larger than the declared type, with the number of bytes being specified by `size`. For example:

```
typedef _Struct_size_bytes_(nSize)
struct MyStruct {
    size_t nSize;
    ...
};
```

The buffer size in bytes of a parameter `pM` of type `MyStruct *` is then taken to be:

```
min(pM->nSize, sizeof(MyStruct))
```

## See Also

- Using SAL Annotations to Reduce C/C++ Code Defects
- Understanding SAL
- Annotating Function Parameters and Return Values
- Annotating Function Behavior
- Annotating Locking Behavior
- Specifying When and Where an Annotation Applies
- Intrinsic Functions
- Best Practices and Examples

# Annotating Locking Behavior

4/16/2019 • 6 minutes to read • [Edit Online](#)

To avoid concurrency bugs in your multithreaded program, always follow an appropriate locking discipline and use SAL annotations.

Concurrency bugs are notoriously hard to reproduce, diagnose, and debug because they are non-deterministic. Reasoning about thread interleaving is difficult at best, and becomes impractical when you are designing a body of code that has more than a few threads. Therefore, it's good practice to follow a locking discipline in your multithreaded programs. For example, obeying a lock order while acquiring multiple locks helps avoid deadlocks, and acquiring the proper guarding lock before accessing a shared resource helps prevent race conditions.

Unfortunately, seemingly simple locking rules can be surprisingly hard to follow in practice. A fundamental limitation in today's programming languages and compilers is that they do not directly support the specification and analysis of concurrency requirements. Programmers have to rely on informal code comments to express their intentions about how they use locks.

Concurrency SAL annotations are designed to help you specify locking side effects, locking responsibility, data guardianship, lock order hierarchy, and other expected locking behavior. By making implicit rules explicit, SAL concurrency annotations provide a consistent way for you to document how your code uses locking rules. Concurrency annotations also enhance the ability of code analysis tools to find race conditions, deadlocks, mismatched synchronization operations, and other subtle concurrency errors.

## General Guidelines

By using annotations, you can state the contracts that are implied by function definitions between implementations (callees) and clients (callers), and express invariants and other properties of the program that can further improve analysis.

SAL supports many different kinds of locking primitives—for example, critical sections, mutexes, spin locks, and other resource objects. Many concurrency annotations take a lock expression as a parameter. By convention, a lock is denoted by the path expression of the underlying lock object.

Some thread ownership rules to keep in mind:

- Spin locks are uncounted locks that have clear thread ownership.
- Mutexes and critical sections are counted locks that have clear thread ownership.
- Semaphores and events are counted locks that do not have clear thread ownership.

## Locking Annotations

The following table lists the locking annotations.

ANNOTATION	DESCRIPTION
<code>_Acquires_exclusive_lock_(expr)</code>	Annotates a function and indicates that in post state the function increments by one the exclusive lock count of the lock object that's named by <code>expr</code> .

ANNOTATION	DESCRIPTION
<code>_Acquires_lock_(expr)</code>	Annotates a function and indicates that in post state the function increments by one the lock count of the lock object that's named by <code>expr</code> .
<code>_Acquires_nonreentrant_lock_(expr)</code>	The lock that's named by <code>expr</code> is acquired. An error is reported if the lock is already held.
<code>_Acquires_shared_lock_(expr)</code>	Annotates a function and indicates that in post state the function increments by one the shared lock count of the lock object that's named by <code>expr</code> .
<code>_Create_lock_level_(name)</code>	A statement that declares the symbol <code>name</code> to be a lock level so that it may be used in the annotations <code>_Has_Lock_level_</code> and <code>_Lock_level_order_</code> .
<code>_Has_lock_kind_(kind)</code>	<p>Annotates any object to refine the type information of a resource object. Sometimes a common type is used for different kinds of resources and the overloaded type is not sufficient to distinguish the semantic requirements among various resources. Here's a list of pre-defined <code>kind</code> parameters:</p> <ul style="list-style-type: none"> <li><code>_Lock_kind_mutex_</code> Lock kind ID for mutexes.</li> <li><code>_Lock_kind_event_</code> Lock kind ID for events.</li> <li><code>_Lock_kind_semaphore_</code> Lock kind ID for semaphores.</li> <li><code>_Lock_kind_spin_lock_</code> Lock kind ID for spin locks.</li> <li><code>_Lock_kind_critical_section_</code> Lock kind ID for critical sections.</li> </ul>
<code>_Has_lock_level_(name)</code>	Annotates a lock object and gives it the lock level of <code>name</code> .
<code>_Lock_level_order_(name1, name2)</code>	A statement that gives the lock ordering between <code>name1</code> and <code>name2</code> .
<code>_Post_same_lock_(expr1, expr2)</code>	Annotates a function and indicates that in post state the two locks, <code>expr1</code> and <code>expr2</code> , are treated as if they are the same lock object.
<code>_Releases_exclusive_lock_(expr)</code>	Annotates a function and indicates that in post state the function decrements by one the exclusive lock count of the lock object that's named by <code>expr</code> .
<code>_Releases_lock_(expr)</code>	Annotates a function and indicates that in post state the function decrements by one the lock count of the lock object that's named by <code>expr</code> .

ANNOTATION	DESCRIPTION
<code>_Releases_nonreentrant_lock_(expr)</code>	The lock that's named by <code>expr</code> is released. An error is reported if the lock is not currently held.
<code>_Releases_shared_lock_(expr)</code>	Annotates a function and indicates that in post state the function decrements by one the shared lock count of the lock object that's named by <code>expr</code> .
<code>_Requires_lock_held_(expr)</code>	Annotates a function and indicates that in pre state the lock count of the object that's named by <code>expr</code> is at least one.
<code>_Requires_lock_not_held_(expr)</code>	Annotates a function and indicates that in pre state the lock count of the object that's named by <code>expr</code> is zero.
<code>_Requires_no_locks_held_</code>	Annotates a function and indicates that the lock counts of all locks that are known to the checker are zero.
<code>_Requires_shared_lock_held_(expr)</code>	Annotates a function and indicates that in pre state the shared lock count of the object that's named by <code>expr</code> is at least one.
<code>_Requires_exclusive_lock_held_(expr)</code>	Annotates a function and indicates that in pre state the exclusive lock count of the object that's named by <code>expr</code> is at least one.

## SAL Intrinsics For Unexposed Locking Objects

Certain lock objects are not exposed by the implementation of the associated locking functions. The following table lists SAL intrinsic variables that enable annotations on functions that operate on those unexposed lock objects.

ANNOTATION	DESCRIPTION
<code>_Global_cancel_spin_lock_</code>	Describes the cancel spin lock.
<code>_Global_critical_region_</code>	Describes the critical region.
<code>_Global_interlock_</code>	Describes interlocked operations.
<code>_Global_priority_region_</code>	Describes the priority region.

## Shared Data Access Annotations

The following table lists the annotations for shared data access.

ANNOTATION	DESCRIPTION
<code>_Guarded_by_(expr)</code>	Annotates a variable and indicates that whenever the variable is accessed, the lock count of the lock object that's named by <code>expr</code> is at least one.

ANNOTATION	DESCRIPTION
<code>_Interlocked_</code>	Annotates a variable and is equivalent to <code>_Guarded_by_( _Global_interlock_ )</code>
<code>_Interlocked_operand_</code>	The annotated function parameter is the target operand of one of the various Interlocked functions. Those operands must have specific additional properties.
<code>_Write_guarded_by_(expr)</code>	Annotates a variable and indicates that whenever the variable is modified, the lock count of the lock object that's named by <code>expr</code> is at least one.

## Smart Lock and RAI Annotations

Smart locks typically wrap native locks and manage their lifetime. The following table lists annotations that can be used with smart locks and RAI coding patterns with support for `move` semantics.

ANNOTATION	DESCRIPTION
<code>_Analysis_assume_smart_lock_acquired_</code>	Tells the analyzer to assume that a smart lock has been acquired. This annotation expects a reference lock type as its parameter.
<code>_Analysis_assume_smart_lock_released_</code>	Tells the analyzer to assume that a smart lock has been released. This annotation expects a reference lock type as its parameter.
<code>_Moves_lock_(target, source)</code>	Describes <code>move constructor</code> operation which transfers lock state from the <code>source</code> object to the <code>target</code> . The <code>target</code> is considered a newly constructed object, so any state it had before is lost and replaced by the <code>source</code> state. The <code>source</code> is also reset to a clean state with no lock counts or aliasing target, but aliases pointing to it remain unchanged.
<code>_Replaces_lock_(target, source)</code>	Describes <code>move assignment operator</code> semantics where the target lock is released before transferring the state from the source. This can be regarded as a combination of <code>_Moves_lock_(target, source)</code> preceded by a <code>_Releases_lock_(target)</code> .
<code>_Swaps_locks_(left, right)</code>	Describes the standard <code>swap</code> behavior which assumes that objects <code>left</code> and <code>right</code> exchange their state. The state exchanged includes lock count and aliasing target, if present. Aliases that point to the <code>left</code> and <code>right</code> objects remain unchanged.

ANNOTATION	DESCRIPTION
<pre>_Detaches_lock_(detached, lock)</pre>	<p>Describes a scenario in which a lock wrapper type allows dissociation with its contained resource. This is similar to how <code>std::unique_ptr</code> works with its internal pointer: it allows programmers to extract the pointer and leave its smart pointer container in a clean state. Similar logic is supported by <code>std::unique_lock</code> and can be implemented in custom lock wrappers. The detached lock retains its state (lock count and aliasing target, if any), while the wrapper is reset to contain zero lock count and no aliasing target, while retaining its own aliases. There's no operation on lock counts (releasing and acquiring). This annotation behaves exactly as <code>_Moves_lock_</code> except that the detached argument should be <code>return</code> rather than <code>this</code>.</p>

## See Also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Specifying When and Where an Annotation Applies](#)
- [Intrinsic Functions](#)
- [Best Practices and Examples](#)
- [Code Analysis Team Blog](#)

# Specifying When and Where an Annotation Applies

2/8/2019 • 2 minutes to read • [Edit Online](#)

When an annotation is conditional, it may require other annotations to specify that to the analyzer. For example, if a function has a variable that can be either synchronous or asynchronous, the function behaves as follows: In the synchronous case it always eventually succeeds, but in the asynchronous case it reports an error if it can't succeed immediately. When the function is called synchronously, checking the result value provides no value to the code analyzer because it would not have returned. However, when the function is called asynchronously and the function result is not checked, a serious error could occur. This example illustrates a situation in which you could use the `_When_` annotation—described later in this article—to enable checking.

## Structural Annotations

To control when and where annotations apply, use the following structural annotations.

ANNOTATION	DESCRIPTION
<code>_At_(expr, anno-list)</code>	<code>expr</code> is an expression that yields an lvalue. The annotations in <code>anno-list</code> are applied to the object that is named by <code>expr</code> . For each annotation in <code>anno-list</code> , <code>expr</code> is interpreted in pre-condition if the annotation is interpreted in pre-condition, and in post-condition if the annotation is interpreted in post-condition.
<code>_At_buffer_(expr, iter, elem-count, anno-list)</code>	<code>expr</code> is an expression that yields an lvalue. The annotations in <code>anno-list</code> are applied to the object that is named by <code>expr</code> . For each annotation in <code>anno-list</code> , <code>expr</code> is interpreted in pre-condition if the annotation is interpreted in precondition, and in post-condition if the annotation is interpreted in post-condition.  <code>iter</code> is the name of a variable that is scoped to the annotation (inclusive of <code>anno-list</code> ). <code>iter</code> has an implicit type <code>long</code> . Identically named variables in any enclosing scope are hidden from evaluation.  <code>elem-count</code> is an expression that evaluates to an integer.
<code>_Group_(anno-list)</code>	The annotations in <code>anno-list</code> are all considered to have any qualifier that applies to the group annotation that is applied to each annotation.

ANNOTATION	DESCRIPTION
<pre>_When_(expr, anno-list)</pre>	<p><code>expr</code> is an expression that can be converted to <code>bool</code>. When it is non-zero (<code>true</code>), the annotations that are specified in <code>anno-list</code> are considered applicable.</p> <p>By default, for each annotation in <code>anno-list</code>, <code>expr</code> is interpreted as using the input values if the annotation is a precondition, and as using the output values if the annotation is a post-condition. To override the default, you can use the <code>_old_</code> intrinsic when you evaluate a post-condition to indicate that input values should be used. <b>Note:</b> Different annotations might be enabled as a consequence of using <code>_When_</code> if a mutable value—for example, <code>*pLength</code>—is involved because the evaluated result of <code>expr</code> in precondition may differ from its evaluated result in post-condition.</p>

## See Also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Intrinsic Functions](#)
- [Best Practices and Examples](#)

# Intrinsic Functions

2/8/2019 • 2 minutes to read • [Edit Online](#)

An expression in SAL can be a C/C++ expression provided that it is an expression that does not have side effects—for example, `++`, `--`, and function calls all have side effects in this context. However, SAL does provide some function-like objects and some reserved symbols that can be used in SAL expressions. These are referred to as *intrinsic functions*.

## General Purpose

The following intrinsic function annotations provide general utility for SAL.

ANNOTATION	DESCRIPTION
<code>_Curr_</code>	A synonym for the object that is currently being annotated. When the <code>_At_</code> annotation is in use, <code>_Curr_</code> is the same as the first parameter to <code>_At_</code> . Otherwise, it is the parameter or the entire function/return value with which the annotation is lexically associated.
<code>_Inexpressible_(expr)</code>	Expresses a situation where the size of a buffer is too complex to represent by using an annotation expression—for example, when it is computed by scanning an input data set and then counting selected members.
<code>_Nullterm_length_(param)</code>	<code>param</code> is the number of elements in the buffer up to but not including a null terminator. It may be applied to any buffer of non-aggregate, non-void type.
<code>_Old_(expr)</code>	When it is evaluated in precondition, <code>_Old_</code> returns the input value <code>expr</code> . When it is evaluated in post-condition, it returns the value <code>expr</code> as it would have been evaluated in precondition.
<code>_Param_(n)</code>	The <code>n</code> th parameter to a function, counting from 1 to <code>n</code> , and <code>n</code> is a literal integral constant. If the parameter is named, this annotation is identical to accessing the parameter by name. <b>Note:</b> <code>n</code> may refer to the positional parameters that are defined by an ellipsis, or may be used in function prototypes where names are not used.
<code>return</code>	The C/C++ reserved keyword <code>return</code> can be used in a SAL expression to indicate the return value of a function. The value is only available in post state; it is a syntax error to use it in pre state.

## String Specific

The following intrinsic function annotations enable manipulation of strings. All four of these functions serve the same purpose: to return the number of elements of the type that is found before a null terminator. The differences are the kinds of data in the elements that are referred to. Note that if you want to specify the length of a null-terminated buffer that is not composed of characters, use the `_Nullterm_length_(param)` annotation from the

previous section.

ANNOTATION	DESCRIPTION
<code>_String_length_(param)</code>	<code>param</code> is the number of elements in the string up to but not including a null terminator. This annotation is reserved for string-of-character types.
<code>strlen(param)</code>	<code>param</code> is the number of elements in the string up to but not including a null terminator. This annotation is reserved for use on character arrays and resembles the C Runtime function <code>strlen()</code> .
<code>wcslen(param)</code>	<code>param</code> is the number of elements in the string up to (but not including) a null terminator. This annotation is reserved for use on wide character arrays and resembles the C Runtime function <code>wcslen()</code> .

## See Also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)
- [Understanding SAL](#)
- [Annotating Function Parameters and Return Values](#)
- [Annotating Function Behavior](#)
- [Annotating Structs and Classes](#)
- [Annotating Locking Behavior](#)
- [Specifying When and Where an Annotation Applies](#)
- [Best Practices and Examples](#)

# Best Practices and Examples (SAL)

2/8/2019 • 5 minutes to read • [Edit Online](#)

Here are some ways to get the most out of the Source Code Annotation Language (SAL) and avoid some common problems.

## \_In\_

If the function is supposed to write to the element, use `_Inout_` instead of `_In_`. This is particularly relevant in cases of automated conversion from older macros to SAL. Prior to SAL, many programmers used macros as comments—macros that were named `IN`, `OUT`, `IN_OUT`, or variants of these names. Although we recommend that you convert these macros to SAL, we also urge you to be careful when you convert them because the code might have changed since the original prototype was written and the old macro might no longer reflect what the code does. Be especially careful about the `OPTIONAL` comment macro because it is frequently placed incorrectly—for example, on the wrong side of a comma.

```
// Incorrect
void Func1(_In_ int *p1)
{
    if (p1 == NULL)
        return;

    *p1 = 1;
}

// Correct
void Func2(_Inout_ PCHAR p1)
{
    if (p1 == NULL)
        return;

    *p1 = 1;
}
```

## \_opt\_

If the caller is not allowed to pass in a null pointer, use `_In_` or `_Out_` instead of `_In_opt_` or `_Out_opt_`. This applies even to a function that checks its parameters and returns an error if it is NULL when it should not be. Although having a function check its parameter for unexpected NULL and return gracefully is a good defensive coding practice, it does not mean that the parameter annotation can be of an optional type (`_*xxx*_opt_`).

```

// Incorrect
void Func1(_Out_opt_ int *p1)
{
    *p = 1;
}

// Correct
void Func2(_Out_ int *p1)
{
    *p = 1;
}

```

## \_Pre\_defensive\_ and \_Post\_defensive\_

If a function appears at a trust boundary, we recommend that you use the `_Pre_defensive_` annotation. The "defensive" modifier modifies certain annotations to indicate that, at the point of call, the interface should be checked strictly, but in the implementation body it should assume that incorrect parameters might be passed. In that case, `_In_ _Pre_defensive_` is preferred at a trust boundary to indicate that although a caller will get an error if it attempts to pass NULL, the function body will be analyzed as if the parameter might be NULL, and any attempts to de-reference the pointer without first checking it for NULL will be flagged. A `_Post_defensive_` annotation is also available, for use in callbacks where the trusted party is assumed to be the caller and the untrusted code is the called code.

## \_Out\_writes\_

The following example demonstrates a common misuse of `_Out_writes_`.

```

// Incorrect
void Func1(_Out_writes_(size) CHAR *pb,
           DWORD size
);

```

The annotation `_Out_writes_` signifies that you have a buffer. It has `cb` bytes allocated, with the first byte initialized on exit. This annotation is not strictly wrong and it is helpful to express the allocated size. However, it does not tell how many elements are initialized by the function.

The next example shows three correct ways to fully specify the exact size of the initialized portion of the buffer.

```

// Correct
void Func1(_Out_writes_to_(size, *pCount) CHAR *pb,
           DWORD size,
           PDWORD pCount
);

void Func2(_Out_writes_all_(size) CHAR *pb,
           DWORD size
);

void Func3(_Out_writes_(size) PSTR pb,
           DWORD size
);

```

## \_Out\_PSTR

The use of `_Out_ PSTR` is almost always wrong. This is interpreted as having an output parameter that points to a character buffer and it is NULL-terminated.

```
// Incorrect
void Func1(_Out_ PSTR pFileName, size_t n);

// Correct
void Func2(_Out_writes_(n) PSTR wszFileName, size_t n);
```

An annotation like `_In_ PCSTR` is common and useful. It points to an input string that has NULL termination because the precondition of `_In_` allows the recognition of a NULL-terminated string.

## `_In_ WCHAR*` p

`_In_ WCHAR* p` says that there is an input pointer `p` that points to one character. However, in most cases, this is probably not the specification that is intended. Instead, what is probably intended is the specification of a NULL-terminated array; to do that, use `_In_ PWSTR`.

```
// Incorrect
void Func1(_In_ WCHAR* wszFileName);

// Correct
void Func2(_In_ PWSTR wszFileName);
```

Missing the proper specification of NULL termination is common. Use the appropriate `STR` version to replace the type, as shown in the following example.

```
// Incorrect
BOOL StrEquals1(_In_ PCHAR p1, _In_ PCHAR p2)
{
    return strcmp(p1, p2) == 0;
}

// Correct
BOOL StrEquals2(_In_ PSTR p1, _In_ PSTR p2)
{
    return strcmp(p1, p2) == 0;
}
```

## `_Out_range_`

If the parameter is a pointer and you want to express the range of the value of the element that is pointed to by the pointer, use `_Deref_out_range_` instead of `_Out_range_`. In the following example, the range of `*pcbFilled` is expressed, not `pcbFilled`.

```

// Incorrect
void Func1(
    _Out_writes_bytes_to_(cbSize, *pcbFilled) BYTE *pb,
    DWORD cbSize,
    _Out_range_(0, cbSize) DWORD *pcbFilled
);

// Correct
void Func2(
    _Out_writes_bytes_to_(cbSize, *pcbFilled) BYTE *pb,
    DWORD cbSize,
    _Deref_out_range_(0, cbSize) _Out_ DWORD *pcbFilled
);

```

`_Deref_out_range_(0, cbSize)` is not strictly required for some tools because it can be inferred from `_Out_writes_to_(cbSize,*pcbFilled)`, but it is shown here for completeness.

## Wrong context in `_When_`

Another common mistake is to use post-state evaluation for preconditions. In the following example, `_Requires_lock_held_` is a precondition.

```

// Incorrect
_WHEN_(return == 0, _Requires_lock_held_(p->cs))
int Func1(_In_ MyData *p, int flag);

// Correct
_WHEN_(flag == 0, _Requires_lock_held_(p->cs))
int Func2(_In_ MyData *p, int flag);

```

The expression `result` refers to a post-state value that is not available in pre-state.

## TRUE in `_Success_`

If the function succeeds when the return value is nonzero, use `return != 0` as the success condition instead of `return == TRUE`. Nonzero does not necessarily mean equivalence to the actual value that the compiler provides for `TRUE`. The parameter to `_Success_` is an expression, and the following expressions are evaluated as equivalent: `return != 0`, `return != false`, `return != FALSE`, and `return` with no parameters or comparisons.

```

// Incorrect
_Success_(return == TRUE, _Acquires_lock_(*lpCriticalSection))
BOOL WINAPI TryEnterCriticalSection(
    _Inout_ LPCRITICAL_SECTION lpCriticalSection
);

// Correct
_Success_(return != 0, _Acquires_lock_(*lpCriticalSection))
BOOL WINAPI TryEnterCriticalSection(
    _Inout_ LPCRITICAL_SECTION lpCriticalSection
);

```

## Reference variable

For a reference variable, the previous version of SAL used the implied pointer as the annotation target and

required the addition of a `_dereference` to annotations that attached to a reference variable. This version uses the object itself and does not require the additional `_Deref_`.

```
// Incorrect
void Func1(
    _Out_writes_bytes_all_(cbSize) BYTE *pb,
    _Deref_ _Out_range_(0, 2) _Out_ DWORD &cbSize
);

// Correct
void Func2(
    _Out_writes_bytes_all_(cbSize) BYTE *pb,
    _Out_range_(0, 2) _Out_ DWORD &cbSize
);
```

## Annotations on return values

The following example shows a common problem in return value annotations.

```
// Incorrect
_Out_opt_ void *MightReturnNullPtr1();

// Correct
_Ret_maybenull_ void *MightReturnNullPtr2();
```

In this example, `_Out_opt_` says that the pointer might be NULL as part of the precondition. However, preconditions cannot be applied to the return value. In this case, the correct annotation is `_Ret_maybenull_`.

## See also

[Using SAL Annotations to Reduce C/C++ Code Defects](#) [Understanding SAL Annotating Function Parameters](#)  
[Annotating Function Behavior](#) [Annotating Structs and Classes](#) [Annotating Locking Behavior](#)  
[Specifying When and Where an Annotation Applies](#) [Intrinsic Functions](#)

# How to: Specify Additional Code Information by Using `_Analysis_assume`

2/8/2019 • 2 minutes to read • [Edit Online](#)

You can provide hints to the code analysis tool for C/C++ code that will help the analysis process and reduce warnings. To provide additional information, use the following function:

```
_Analysis_assume( expr )
```

`expr` - any expression that is assumed to evaluate to true.

The code analysis tool assumes that the condition represented by the expression is true at the point where the function appears and remains true until expression is altered, for example, by assignment to a variable.

## NOTE

`_Analysis_assume` does not impact code optimization. Outside the code analysis tool, `_Analysis_assume` is defined as a no-op.

## Example

The following code uses `_Analysis_assume` to correct the code analysis warning [C6388](#):

```
#include<windows.h>
#include<codeanalysis\sourceannotations.h>

using namespace vc_attributes;

// calls free and sets ch to null
void FreeAndNull(char* ch);

//requires pc to be null
void f([Pre(Null=Yes)] char* pc);

void test( )
{
    char *pc = (char*)malloc(5);
    FreeAndNull(pc);
    _Analysis_assume(pc == NULL);
    f(pc);
}
```

## See Also

[\\_assume](#)

# Use rule sets to group code analysis rules

3/27/2019 • 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 [static analysis of managed code](#), [analysis of C++ code](#), and [Roslyn analyzers](#).

## 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](#)

# Rule sets for Roslyn analyzers

3/12/2019 • 2 minutes to read • [Edit Online](#)

Predefined rule sets are included with some NuGet analyzer packages. For example, the rule sets that are included with the [Microsoft.CodeAnalysis.FxCopAnalyzers](#) NuGet analyzer package (starting in version 2.6.2) enable or disable rules based on their category, such as security, naming, or performance. Using rule sets makes it easy to quickly see only those rule violations that pertain to a particular category of rule.

If you're migrating from legacy "FxCop" static code analysis to Roslyn analyzers, these rule sets enable you to continue using the same rule configurations you used previously.

## Use analyzer rule sets

After you [install a NuGet analyzer package](#), locate the predefined rule set in its *rulesets* directory, for example `%USERPROFILE%\.nuget\packages\microsoft.codequality.analyzers<version>\rulesets`. From there, you can drag and drop, or copy and paste, one or more of the rulesets to your Visual Studio project in **Solution Explorer**.

To make a rule set the active rule set for analysis, 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. Now you only see rule violations for those rules that are enabled in the selected rule set.

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**.

## Available rule sets

The predefined analyzer rule sets include three rulesets that affect all the rules in the package—one that enables them all, one that disables them all, and one that honors each rule's default severity and enablement settings:

- AllRulesEnabled.ruleset
- AllRulesDisabled.ruleset
- AllRulesDefault.ruleset

Additionally, there are two rule sets for each category of rules in the package, such as performance or security. One rule set enables all rules for the category, and one rule set honors the default severity and enablement settings for each rule in the category.

The [Microsoft.CodeAnalysis.FxCopAnalyzers](#) NuGet analyzer package includes rule sets for the following categories, which match the rule sets available for legacy "FxCop" static code analysis:

- design
- documentation
- maintainability
- naming
- performance
- reliability
- security
- usage

## See also

- [Analyzers FAQ](#)
- [Overview of .NET Compiler Platform analyzers](#)
- [Install analyzers](#)
- [Use analyzers](#)
- [Use rule sets to group code analysis rules](#)

# How to: Configure Code Analysis for a Managed Code Project

2/8/2019 • 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.

## TIP

For information about how to configure a rule set for ASP.NET web applications, see [How to: Configure Code Analysis for an ASP.NET web Application](#).

## To 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 the **Enable Code Analysis on Build** check box. You can also run code analysis manually by selecting **Analyze > Run Code Analysis > Run Code Analysis on <projectname>**.
4. By default, code analysis does not report warnings from code that is automatically generated by external tools. 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, without having it overwritten.

5. In the **Run this rule set** 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: Configure Code Analysis for an ASP.NET web Application](#)

# How to: Configure Code Analysis for an ASP.NET Web Application

2/8/2019 • 2 minutes to read • [Edit Online](#)

In Visual Studio, you can select from a list of Code Analysis *rule sets* to apply to ASP.NET web application. The default rule set is the Microsoft Minimum Recommended Rules. You can select another rule set to apply to the web site.

1. Select the web site in **Solution Explorer**.
2. On the **Analyze** menu, click **Configure Code Analysis for Web Site**.
3. If you selected the solution and the solution has more than one project, select the build configuration and target operating system from the **Configuration** and **Platform** lists.
4. For each project in the solution, click the **Rule Set** column, and then click the name of the rule set to run.
5. By default, code analysis is run on all projects in the solution. To disable or enable code analysis for a particular project, follow these steps:
  - a. Right-click the project name and then click Properties.
  - b. Check or clear the **Enable Code Analysis** check box. You can also run code analysis manually by selecting **Run Code Analysis on Web Site** from the **Analyze** menu.
6. In the **Run this rule set** drop-down list, follow these steps:
  - Select the rule set that you want to use.
  - Select <**Browse**> to specify an existing custom rule set that is not in the list.
  - Define a [custom rule set](#).

# Use Rule Sets to Specify the C++ Rules to Run

3/4/2019 • 3 minutes to read • [Edit Online](#)

In Visual Studio, you can create and modify a custom *rule set* to meet specific project needs associated with code analysis. The default rule sets are stored in `%VSINSTALLDIR%\Team Tools\Static Analysis Tools\Rule Sets`.

**Visual Studio 2017 version 15.7 and later** You can create custom rule sets using any text editor and apply them in command line builds no matter what build system you are using. For more information, see [/analyze:ruleset](#).

To create a custom C++ rule set in Visual Studio, a C/C++ project must be open in the Visual Studio IDE. You then open a standard rule set in the rule set editor and then add or remove specific rules and optionally change the action that occurs when code analysis determines that a rule has been violated.

To create a new custom rule set, you save it by using a new file name. The custom rule set is automatically assigned to the project.

## To create a custom rule from a single existing rule set

1. In Solution Explorer, open the shortcut menu for the project and then choose **Properties**.
2. On the **Properties** tab, choose **Code Analysis**.
3. In the **Rule Set** drop-down list, do one of the following:
  - Choose the rule set that you want to customize.
  - or -
  - Choose <**Browse...**> to specify an existing rule set that is not in the list.
4. Choose **Open** to display the rules in the rule set editor.

## To modify a rule set in the rule set editor

- To change the display name of the rule set, on the **View** menu, choose **Properties Window**. Enter the display name in the **Name** box. Notice that the display name can differ from the file name.
- To add all the rules of the group to a custom rule set, select the check box of the group. To remove all the rules of the group, clear the check box.
- To add a specific rule to the custom rule set, select the check box of the rule. To remove the rule from the rule set, clear the check box.
- To change the action taken when a rule is violated in a code analysis, choose the **Action** field for the rule and then choose one of the following values:

**Warn** - generates a warning.

**Error** - generates an error.

**None** - disables the rule. This action is the same as removing the rule from the rule set.

## To group, filter, or change the fields in the rule set editor by using the rule set editor toolbar

- To expand the rules in all groups, choose **Expand All**.
- To collapse the rules in all groups, choose **Collapse All**.
- To change the field that rules are grouped by, choose the field from the **Group By** list. To display the rules ungrouped, choose <**None**>.
- To add or remove fields in rule columns, choose **Column Options**.
- To hide rules that do not apply to the current solution, choose **Hide rules that do not apply to the current solution**.
- To switch between showing and hiding rules that are assigned the Error action, choose **Show rules that can generate Code Analysis errors**.
- To switch between showing and hiding rules that are assigned the Warning action, choose **Show rules that can generate Code Analysis warnings**.
- To switch between showing and hiding rules that are assigned the **None** action, choose **Show rules that are not enabled**.
- To add or remove Microsoft default rule sets to the current rule set, choose **Add or remove child rule sets**.

## To create a rule set in a text editor

You can create a custom rule set in a text editor, store it in any location with a `.ruleset` extension, and apply it with the `/analyzer.ruleset` compiler option.

The following example shows a basic rule set file that you can use as a starting point:

```
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="New Rule Set" Description=" " ToolsVersion="15.0">
  <Rules AnalyzerId="Microsoft.Analyzers.NativeCodeAnalysis" RuleNamespace="Microsoft.Rules.Native">
    <Rule Id="C6001" Action="Warning" />
    <Rule Id="C26494" Action="Warning" />
  </Rules>
</RuleSet>
```

```
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="New Rule Set" Description=" " ToolsVersion="16.0">
  <Rules AnalyzerId="Microsoft.Analyzers.NativeCodeAnalysis" RuleNamespace="Microsoft.Rules.Native">
    <Rule Id="C6001" Action="Warning" />
    <Rule Id="C26494" Action="Warning" />
  </Rules>
</RuleSet>
```

# Customize a rule set

2/8/2019 • 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

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:
  - 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.

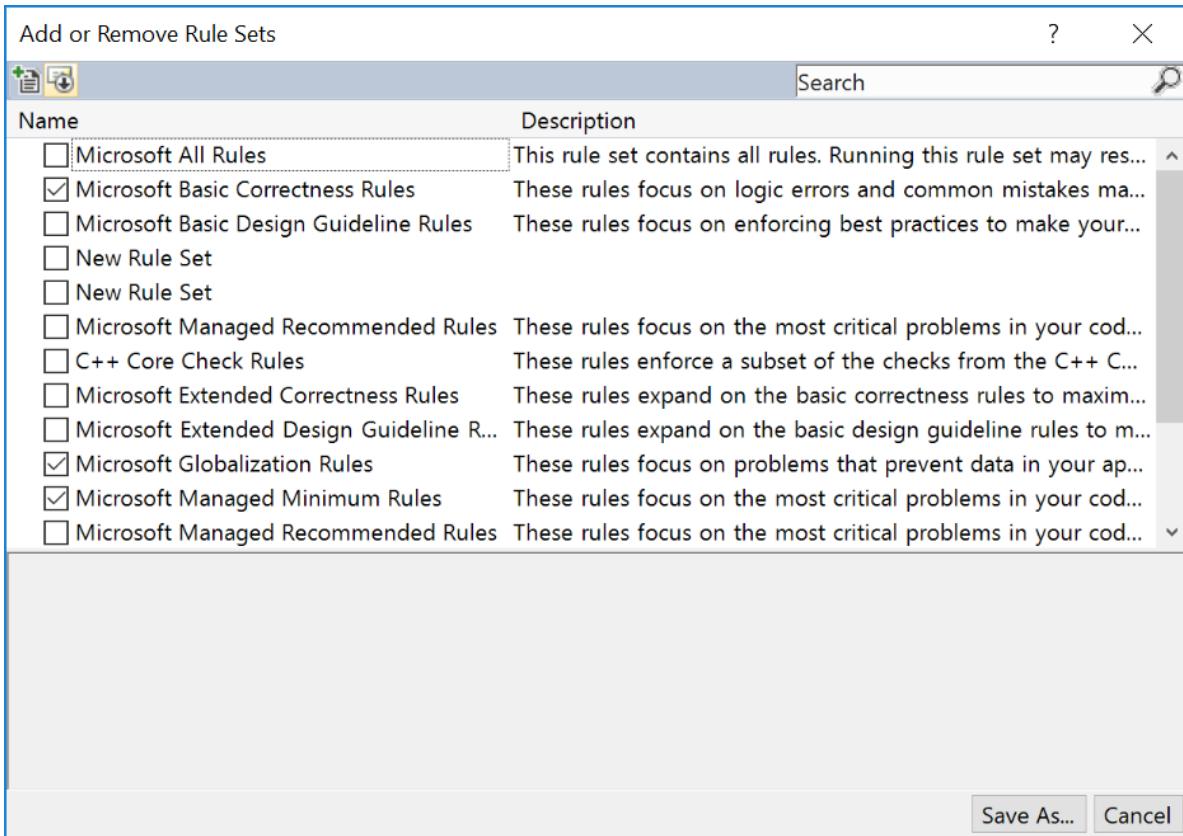
You can also create a new rule set file from the **New File** dialog:

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

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**.
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

2/8/2019 • 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 <b>Error List</b> and also at build time.
Error	Generates an error in the <b>Error List</b> and also at build time.
Info	Generates a message in the <b>Error List</b> .
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
<b>Expand All</b>	Shows the rules in all groups.
<b>Collapse All</b>	Hides the rules in all groups.
<b>Group By</b>	Specifies the field by which rules are grouped. Click < <b>None</b> > to show the rules without groups.

TOOLBAR CONTROL	DESCRIPTION
<b>Column Options</b>	Specifies the rule fields to display.
<b>Hide rules that do not apply to the current solution</b>	Shows or hides rules that are not of the same Target Type as the solution.
<b>Show rules that can generate Code Analysis errors</b>	Shows or hides rules that are assigned the Error action.
<b>Show rules that can generate Code Analysis warnings</b>	Shows or hides rules that are assigned the Warning action.
<b>Show rules that are not enabled</b>	Shows or hides rules that are assigned the None action.
<b>Add or remove child rule sets</b>	Adds or removes the rules in the selected rule sets.
<b>Search rules</b>	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
<b>ID</b>	The identifier of the rule.
<b>Category</b>	In addition to their membership in rule sets, code analysis rules are also grouped by category. For more information, see <a href="#">Code analysis warnings</a> .
<b>Name</b>	The title of the rule.
<b>Namespace</b>	The namespace of the rule.
<b>Target Type</b>	Indicates whether the rule is for native, managed, or database code.
<b>Action</b>	The action taken when the rule is violated in a code analysis run. You can edit the <b>Action</b> field.
<b>Source Rule Sets</b>	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](#)

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

2/8/2019 • 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
<b>Enforce check-in to only contain files that are part of current solution.</b>	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
<b>Enforce C/C++ Code Analysis (/analyze)</b>	Requires that all C or C++ projects be built with the /analyze compiler option to run code analysis before they can be checked in.
<b>Enforce Code Analysis for Managed Code</b>	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

- [Create and use code analysis check-in policies](#)

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

2/8/2019 • 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:

\$/  /

**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 the **Enable Code Analysis on Build (defines CODE\_ANALYSIS constant)** check box.
6. To ignore code in components from other companies, select the **Suppress results from generated code** check box.
7. In the **Run this rule set** list, click <**Browse...>**.
8. Specify the local version of the check-in policy rule set file.

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

2/8/2019 • 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

2/8/2019 • 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 on the Code Analysis page of the code project settings. The Microsoft standard rule sets are automatically installed with Visual Studio and 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, click 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...>**.
5. In the **Open** dialog box, specify the check-in policy rule set file.

## To create a custom rule set for a code project

1. Follow one of the procedures earlier in this topic to select the check-in policy of the Azure DevOps project on the Code Analysis page of the project settings dialog box.
2. Click **Open**.

3. Add or remove rules by using the [rule set editor](#).
4. Save the modified rule set to a .ruleset file on the local computer or to a UNC path.
5. Open the properties dialog box for the code project, and display the **Code Analysis** page.
6. In the **Run this rule set** list, click <**Browse...**>.
7. In the **Open** dialog box, specify the rule set file.

# Version Compatibility for Code Analysis Check-In Policies

2/8/2019 • 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.

# How to: Generate code metrics data

4/8/2019 • 6 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.CodeAnalysis.Analyzers" RuleNamespace="Microsoft.CodeAnalysis.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** column.

#### 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 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>
                    </Namespace>
                </Namespaces>
                <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>
            </Namespace>
        </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 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

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

2/8/2019 • 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

When you configure static code analysis for managed code projects in Visual Studio, you can choose from a list of built-in *rule sets*. You can either use one of these built-in rule sets, or you can [customize a rule set](#) to fit your project requirements.

The topics in this section describe the built-in rule sets and the rules (or warnings) they contain.

## NOTE

The rule sets in this section pertain to static code analysis. For information about rule sets available for Roslyn analyzer packages, see [Use rule sets with Roslyn analyzers](#).

- [All Rules rule set](#)
- [Basic Correctness Rules rule set for managed code](#)
- [Basic Design Guideline Rules rule set for managed code](#)
- [Extended Correctness Rules rule set for managed code](#)
- [Extended Design Guidelines Rules rule set for managed code](#)
- [Globalization Rules rule set for managed code](#)
- [Managed Minimum Rules rule set for managed code](#)
- [Managed Recommended Rules rule set for managed code](#)
- [Mixed Minimum Rules rule set](#)
- [Mixed Recommended Rules rule set](#)
- [Native Minimum Rules rule set](#)
- [Native Recommended Rules rule set](#)
- [Security Rules rule set for managed code](#)

# All Rules rule set

2/8/2019 • 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

2/8/2019 • 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 Minimum Recommended Rules rule set. For more information, see [Managed Recommended Rules rule set for managed code](#). You should include this rule set to expand on the list of warnings that the minimum recommended rules report.

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

Rule	Description
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
CA2146	Types must be at least as critical as their base types and interfaces
CA2147	Transparent methods may not use security asserts

RULE	DESCRIPTION
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
CA1806	Do not ignore method results
CA1816	Call GC.SuppressFinalize correctly

RULE	DESCRIPTION
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-CLSCCompliant 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

2/8/2019 • 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 Microsoft Minimum Recommended Rules rule set. For a list of the minimum rules, see [Managed Recommended Rules rule set for managed code](#).

The following table describes all of the rules in the Microsoft Basic 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

# Extended Correctness Rules rule set for managed code

2/8/2019 • 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 Microsoft Basic Correctness Rules rule set. The Basic Correctness Rules include the rules that are in the Microsoft Minimum Recommended Rules rule set. For more information see [Basic Correctness Rules rule set for managed code](#) and [Managed Recommended Rules rule set for managed code](#)

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

RULE	DESCRIPTION
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
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

RULE	DESCRIPTION
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
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

RULE	DESCRIPTION
CA1308	Normalize strings to uppercase
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-CLSCCompliant 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

RULE	DESCRIPTION
CA2239	Provide deserialization methods for optional fields
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

RULE	DESCRIPTION
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
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

2/8/2019 • 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 of the Microsoft Basic Design Guideline Rules. The Basic Design Guideline Rules include all of the Microsoft Minimum Recommended Rules. For more information, see [Basic Design Guideline Rules rule set for managed code](#) and [Managed Recommended Rules rule set for managed code](#)

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

RULE	DESCRIPTION
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
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

Rule	Description
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
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

Rule	Description
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
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

Rule	Description
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
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

RULE	DESCRIPTION
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
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

RULE	DESCRIPTION
CA2234	Pass System.Uri objects instead of strings
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

RULE	DESCRIPTION
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
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

You can 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

2/8/2019 • 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
<a href="#">CA1001</a>	Types that own disposable fields should be disposable
<a href="#">CA1821</a>	Remove empty finalizers
<a href="#">CA2213</a>	Disposable fields should be disposed
<a href="#">CA2231</a>	Overload operator equals on overriding ValueType.Equals

# Managed Recommended Rules rule set for managed code

2/8/2019 • 2 minutes to read • [Edit Online](#)

You can 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. You should 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
CA1900	Value type fields should be portable

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

RULE	DESCRIPTION
<a href="#">CA2147</a>	Transparent methods may not use security asserts
<a href="#">CA2149</a>	Transparent methods must not call into native code
<a href="#">CA2200</a>	Rethrow to preserve stack details
<a href="#">CA2202</a>	Do not dispose objects multiple times
<a href="#">CA2207</a>	Initialize value type static fields inline
<a href="#">CA2212</a>	Do not mark serviced components with WebMethod
<a href="#">CA2213</a>	Disposable fields should be disposed
<a href="#">CA2214</a>	Do not call overridable methods in constructors
<a href="#">CA2216</a>	Disposable types should declare finalizer
<a href="#">CA2220</a>	Finalizers should call base class finalizer
<a href="#">CA2229</a>	Implement serialization constructors
<a href="#">CA2231</a>	Overload operator equals on overriding ValueType.Equals
<a href="#">CA2232</a>	Mark Windows Forms entry points with STAThread
<a href="#">CA2235</a>	Mark all non-serializable fields
<a href="#">CA2236</a>	Call base class methods on ISerializable types
<a href="#">CA2237</a>	Mark ISerializable types with SerializableAttribute
<a href="#">CA2238</a>	Implement serialization methods correctly
<a href="#">CA2240</a>	Implement ISerializable correctly
<a href="#">CA2241</a>	Provide correct arguments to formatting methods
<a href="#">CA2242</a>	Test for NaN correctly

# Mixed Minimum Rules rule set

2/8/2019 • 5 minutes to read • [Edit Online](#)

The Microsoft Mixed Minimum Rules focus on the most critical problems in your C++ projects that support the Common Language Runtime, including potential security holes and application crashes. You should 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
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
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
C28211	Static context name expected for SAL_context

RULE	DESCRIPTION
C28212	Pointer expression expected for annotation
C28213	The <code>_Use_decl_annotations_</code> 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	<code>_yes</code> , <code>_no</code> , or <code>_maybe</code> 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

RULE	DESCRIPTION
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

2/8/2019 • 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. You should 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
C6226	Invalid HRESULT Assignment To -1

Rule	Description
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
C6280	Memory Allocation-Deallocation Mismatch

RULE	DESCRIPTION
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
C6312	Exception Continue Execution Loop

RULE	DESCRIPTION
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
C6509	Return Used On Precondition

Rule	Description
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
C26101	Failing to use interlocked operation properly

RULE	DESCRIPTION
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 <code>_Function_class_</code> 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
C28164	The argument is being passed to a function that expects a pointer to an object (not a pointer to a pointer)

RULE	DESCRIPTION
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
C28220	Integer expression expected for annotation the parameter in the annotation

RULE	DESCRIPTION
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
C28243	The annotation for function on parameter requires more dereferences than the actual type annotated allows

RULE	DESCRIPTION
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
C28287	For function, syntax Error in _At_() annotation (unrecognized parameter name)

RULE	DESCRIPTION
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
CA1063	Implement IDisposable correctly
CA1065	Do not raise exceptions in unexpected locations

RULE	DESCRIPTION
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
CA2126	Type link demands require inheritance demands
CA2131	Security critical types may not participate in type equivalence

Rule	Description
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
CA2236	Call base class methods on ISerializable types
CA2237	Mark ISerializable types with SerializableAttribute

RULE	DESCRIPTION
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

2/8/2019 • 5 minutes to read • [Edit Online](#)

The Microsoft Native Minimum Rules focus on the most critical problems in your native code, including potential security holes and application crashes. You should include this rule set in any custom rule set you create for your 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

RULE	DESCRIPTION
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
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)

RULE	DESCRIPTION
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	<code>_at</code> 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.

RULE	DESCRIPTION
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
C28287	For function, syntax Error in _At_0 annotation (unrecognized parameter name)
C28288	For function, syntax Error in _At_0 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.

RULE	DESCRIPTION
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.

# Native Recommended Rules rule set

2/8/2019 • 8 minutes to read • [Edit Online](#)

The Native Recommended Rules focus on the most critical and common problems in your native code, including potential security holes and application crashes. You should include this rule set in any custom rule set you create for your 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
C6226	Invalid HRESULT Assignment To -1

Rule	Description
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
C6280	Memory Allocation-Deallocation Mismatch

RULE	DESCRIPTION
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
C6312	Exception Continue Execution Loop

RULE	DESCRIPTION
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
C6509	Return Used On Precondition

Rule	Description
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
C26101	Failing to use interlocked operation properly

RULE	DESCRIPTION
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_RAI_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 <code>_Function_class_</code> 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

RULE	DESCRIPTION
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
C28218	For function parameter, the annotation's parameter does not match that found at file

RULE	DESCRIPTION
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
C28240	The annotation on parameter has param2 but no param1

RULE	DESCRIPTION
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	<code>dynamic_cast&lt;&gt;()</code> 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 <code>_Macro_value_</code> 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

RULE	DESCRIPTION
C28286	For function, syntax error near the end
C28287	For function, syntax Error in _At_0 annotation (unrecognized parameter name)
C28288	For function, syntax Error in _At_0 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

4/9/2019 • 2 minutes to read • [Edit Online](#)

You should include the Microsoft Security Rules rule set to maximize the number of potential security issues that are reported.

RULE	DESCRIPTION
CA2100	Review SQL queries for security vulnerabilities
CA2102	Catch non-CLSCCompliant 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
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

# Code Analysis for Managed Code Warnings

2/8/2019 • 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

<a href="#">Warnings By CheckId</a>	Lists all warnings by CheckId
<a href="#">Cryptography Warnings</a>	Warnings that support safer libraries and applications through the correct use of cryptography.

<a href="#">Design Warnings</a>	Warnings that support correct library design as specified by the .NET Framework Design Guidelines.
<a href="#">Globalization Warnings</a>	Warnings that support world-ready libraries and applications.
<a href="#">Interoperability Warnings</a>	Warnings that support interaction with COM clients.
<a href="#">Maintainability Warnings</a>	Warnings that support library and application maintenance.
<a href="#">Mobility Warnings</a>	Warnings that support efficient power usage.
<a href="#">Naming Warnings</a>	Warnings that support adherence to the naming conventions of the .NET Framework Design Guidelines.
<a href="#">Performance Warnings</a>	Warnings that support high-performance libraries and applications.
<a href="#">Portability Warnings</a>	Warnings that support portability across different platforms.
<a href="#">Reliability Warnings</a>	Warnings that support library and application reliability, such as correct memory and thread usage.
<a href="#">Security Warnings</a>	Warnings that support safer libraries and applications.
<a href="#">Usage Warnings</a>	Warnings that support appropriate usage of the .NET Framework.
<a href="#">Code Analysis Policy Errors</a>	Errors that occur if the code analysis policy is not satisfied at check-in.

# Code analysis warnings for managed code by CheckId

2/8/2019 • 50 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	<a href="#">CA1000: Do not declare static members on generic types</a>	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	<a href="#">CA1001: Types that own disposable fields should be disposable</a>	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	<a href="#">CA1002: Do not expose generic lists</a>	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	<a href="#">CA1003: Use generic event handler instances</a>	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	<a href="#">CA1004: Generic methods should provide type parameter</a>	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	<a href="#">CA1005: Avoid excessive parameters on generic types</a>	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 <code>List&lt;T&gt;</code> , and in certain cases that have two type parameters, as in <code>Dictionary&lt; TKey, TValue &gt;</code> . However, if more than two type parameters exist, the difficulty becomes too great for most users.
CA1006	<a href="#">CA1006: Do not nest generic types in member signatures</a>	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	<a href="#">CA1007: Use generics where appropriate</a>	An externally visible method contains a reference parameter of type <code>System.Object</code> . 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	<a href="#">CA1008: Enums should have zero value</a>	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 <code>FlagsAttribute</code> 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	<a href="#">CA1009: Declare event handlers correctly</a>	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	<a href="#">CA1010: Collections should implement generic interface</a>	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	<a href="#">CA1011: Consider passing base types as parameters</a>	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	<a href="#">CA1012: Abstract types should not have constructors</a>	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	<a href="#">CA1013: Overload operator equals on overloading add and subtract</a>	A public or protected type implements the addition or subtraction operators without implementing the equality operator.
CA1014	<a href="#">CA1014: Mark assemblies with CLSCompliantAttribute</a>	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 <a href="#">CLSCompliantAttribute</a> . If this attribute is not present on an assembly, the assembly is not compliant.

CHECKID	WARNING	DESCRIPTION
CA1016	<a href="#">CA1016: Mark assemblies with AssemblyVersionAttribute</a>	The .NET Framework 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	<a href="#">CA1017: Mark assemblies with ComVisibleAttribute</a>	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	<a href="#">CA1018: Mark attributes with AttributeUsageAttribute</a>	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	<a href="#">CA1019: Define accessors for attribute arguments</a>	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	<a href="#">CA1020: Avoid namespaces with few types</a>	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	<a href="#">CA1021: Avoid out parameters</a>	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	<a href="#">CA1023: Indexers should not be multidimensional</a>	Indexers (that is, indexed properties) should use a single index. Multidimensional indexers can significantly reduce the usability of the library.
CA1024	<a href="#">CA1024: Use properties where appropriate</a>	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	<a href="#">CA1025: Replace repetitive arguments with params array</a>	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	<a href="#">CA1026: Default parameters should not be used</a>	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	<a href="#">CA1027: Mark enums with FlagsAttribute</a>	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	<a href="#">CA1028: Enum storage should be Int32</a>	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	<a href="#">CA1030: Use events where appropriate</a>	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	<a href="#">CA1031: Do not catch general exception types</a>	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	<a href="#">CA1032: Implement standard exception constructors</a>	Failure to provide the full set of constructors can make it difficult to correctly handle exceptions.
CA1033	<a href="#">CA1033: Interface methods should be callable by child types</a>	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	<a href="#">CA1034: Nested types should not be visible</a>	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	<a href="#">CA1035: ICollection implementations have strongly typed members</a>	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	<a href="#">CA1036: Override methods on comparable types</a>	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	<a href="#">CA1038: Enumerators should be strongly typed</a>	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	<a href="#">CA1039: Lists are strongly typed</a>	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	<a href="#">CA1040: Avoid empty interfaces</a>	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	<a href="#">CA1041: Provide ObsoleteAttribute message</a>	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	<a href="#">CA1043: Use integral or string argument for indexers</a>	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	<a href="#">CA1044: Properties should not be write only</a>	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	<a href="#">CA1045: Do not pass types by reference</a>	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	<a href="#">CA1046: Do not overload operator equals on reference types</a>	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	<a href="#">CA1047: Do not declare protected members in sealed types</a>	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	<a href="#">CA1048: Do not declare virtual members in sealed types</a>	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	<a href="#">CA1049: Types that own native resources should be disposable</a>	Types that allocate unmanaged resources should implement <code>IDisposable</code> to enable callers to release those resources on demand and to shorten the lifetimes of the objects that hold the resources.
CA1050	<a href="#">CA1050: Declare types in namespaces</a>	Types are declared in namespaces to prevent name collisions and as a way to organize related types in an object hierarchy.
CA1051	<a href="#">CA1051: Do not declare visible instance fields</a>	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	<a href="#">CA1052: Static holder types should be sealed</a>	A public or protected type contains only static members and is not declared by using the <code>sealed</code> (C# Reference) ( <code>NotInheritable</code> ) modifier. A type that is not meant to be inherited should be marked by using the <code>sealed</code> modifier to prevent its use as a base type.
CA1053	<a href="#">CA1053: Static holder types should not have constructors</a>	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	<a href="#">CA1054: URI parameters should not be strings</a>	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	<a href="#">CA1055: URI return values should not be strings</a>	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	<a href="#">CA1056: URI properties should not be strings</a>	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	<a href="#">CA1057: String URI overloads call System.Uri overloads</a>	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	<a href="#">CA1058: Types should not extend certain base types</a>	An externally visible type extends certain base types. Use one of the alternatives.
CA1059	<a href="#">CA1059: Members should not expose certain concrete types</a>	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	<a href="#">CA1060: Move P/Invokes to NativeMethods class</a>	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	<a href="#">CA1061: Do not hide base class methods</a>	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	<a href="#">CA1062: Validate arguments of public methods</a>	All reference arguments that are passed to externally visible methods should be checked against null.
CA1063	<a href="#">CA1063: Implement IDisposable correctly</a>	All IDisposable types should implement the Dispose pattern correctly.
CA1064	<a href="#">CA1064: Exceptions should be public</a>	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 <a href="#">Exception</a> , <a href="#">SystemException</a> , or <a href="#">ApplicationException</a> , the external code will not have sufficient information to know what to do with the exception.
CA1065	<a href="#">CA1065: Do not raise exceptions in unexpected locations</a>	A method that is not expected to throw exceptions throws an exception.
CA1300	<a href="#">CA1300: Specify MessageBoxOptions</a>	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	<a href="#">CA1301: Avoid duplicate accelerators</a>	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	<a href="#">CA1302: Do not hardcode locale specific strings</a>	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.

CHECKID	WARNING	DESCRIPTION
CA1303	<a href="#">CA1303: Do not pass literals as localized parameters</a>	An externally visible method passes a string literal as a parameter to a constructor or method in the .NET Framework class library, and that string should be localizable.
CA1304	<a href="#">CA1304: Specify CultureInfo</a>	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	<a href="#">CA1305: Specify IFormatProvider</a>	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	<a href="#">CA1306: Set locale for data types</a>	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	<a href="#">CA1307: Specify StringComparison</a>	A string comparison operation uses a method overload that does not set a StringComparison parameter.
CA1308	<a href="#">CA1308: Normalize strings to uppercase</a>	Strings should be normalized to uppercase. A small group of characters cannot make a round trip when they are converted to lowercase.

CHECKID	WARNING	DESCRIPTION
CA1309	<a href="#">CA1309: Use ordinal StringComparison</a>	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	<a href="#">CA1400: P/Invoke entry points should exist</a>	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	<a href="#">CA1401: P/Invokes should not be visible</a>	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	<a href="#">CA1402: Avoid overloads in COM visible interfaces</a>	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	<a href="#">CA1403: Auto layout types should not be COM visible</a>	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 the .NET Framework, which will break COM clients that expect a specific layout.
CA1404	<a href="#">CA1404: Call GetLastError immediately after P/Invoke</a>	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	<a href="#">CA1405: COM visible type base types should be COM visible</a>	A COM-visible type derives from a type that is not COM-visible.
CA1406	<a href="#">CA1406: Avoid Int64 arguments for Visual Basic 6 clients</a>	Visual Basic 6 COM clients cannot access 64-bit integers.

CHECKID	WARNING	DESCRIPTION
CA1407	<a href="#">CA1407: Avoid static members in COM visible types</a>	COM does not support static methods.
CA1408	<a href="#">CA1408: Do not use AutoDual ClassInterfaceType</a>	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	<a href="#">CA1409: Com visible types should be creatable</a>	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	<a href="#">CA1410: COM registration methods should be matched</a>	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	<a href="#">CA1411: COM registration methods should not be visible</a>	A method marked by using the System.Runtime.InteropServices.ComRegisterFunctionAttribute attribute or the System.Runtime.InteropServices.ComUnregisterFunctionAttribute attribute is externally visible.
CA1412	<a href="#">CA1412: Mark ComSource Interfaces as IDispatch</a>	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	<a href="#">CA1413: Avoid non-public fields in COM visible value types</a>	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	<a href="#">CA1414: Mark boolean P/Invoke arguments with MarshalAs</a>	The Boolean data type has multiple representations in unmanaged code.

CHECKID	WARNING	DESCRIPTION
CA1415	<a href="#">CA1415: Declare P/Invokes correctly</a>	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.
CA1500	<a href="#">CA1500: Variable names should not match field names</a>	An instance method declares a parameter or a local variable whose name matches an instance field of the declaring type, leading to errors.
CA1501	<a href="#">CA1501: Avoid excessive inheritance</a>	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	<a href="#">CA1502: Avoid excessive complexity</a>	This rule measures the number of linearly independent paths through the method, which is determined by the number and complexity of conditional branches.
CA1504	<a href="#">CA1504: Review misleading field names</a>	The name of an instance field starts with "s_", or the name of a static (Shared in Visual Basic) field starts with "m_".
CA1505	<a href="#">CA1505: Avoid unmaintainable code</a>	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	<a href="#">CA1506: Avoid excessive class coupling</a>	This rule measures class coupling by counting the number of unique type references that a type or method contains.
CA1600	<a href="#">CA1600: Do not use idle process priority</a>	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	<a href="#">CA1601: Do not use timers that prevent power state changes</a>	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	<a href="#">CA1700: Do not name enum values 'Reserved'</a>	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	<a href="#">CA1701: Resource string compound words should be cased correctly</a>	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	<a href="#">CA1702: Compound words should be cased correctly</a>	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	<a href="#">CA1703: Resource strings should be spelled correctly</a>	A resource string contains one or more words that are not recognized by the Microsoft spelling checker library.
CA1704	<a href="#">CA1704: Identifiers should be spelled correctly</a>	The name of an externally visible identifier contains one or more words that are not recognized by the Microsoft spelling checker library.
CA1707	<a href="#">CA1707: Identifiers should not contain underscores</a>	By convention, identifier names do not contain the underscore (_) character. This rule checks namespaces, types, members, and parameters.
CA1708	<a href="#">CA1708: Identifiers should differ by more than case</a>	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	<a href="#">CA1709: Identifiers should be cased correctly</a>	By convention, parameter names use camel casing and namespace, type, and member names use Pascal casing.
CA1710	<a href="#">CA1710: Identifiers should have correct suffix</a>	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	<a href="#">CA1711: Identifiers should not have incorrect suffix</a>	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	<a href="#">CA1712: Do not prefix enum values with type name</a>	Names of enumeration members are not prefixed by using the type name because development tools are expected to provide type information.
CA1713	<a href="#">CA1713: Events should not have before or after prefix</a>	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	<a href="#">CA1714: Flags enums should have plural names</a>	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	<a href="#">CA1715: Identifiers should have correct prefix</a>	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	<a href="#">CA1716: Identifiers should not match keywords</a>	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	<a href="#">CA1717: Only FlagsAttribute enums should have plural names</a>	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	<a href="#">CA1719: Parameter names should not match member names</a>	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	<a href="#">CA1720: Identifiers should not contain type names</a>	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	<a href="#">CA1721: Property names should not match get methods</a>	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	<a href="#">CA1722: Identifiers should not have incorrect prefix</a>	By convention, only certain programming elements have names that begin with a specific prefix.
CA1724	<a href="#">CA1724: Type Names Should Not Match Namespaces</a>	Type names should not match the names of namespaces that are defined in the .NET Framework class library. Violating this rule can reduce the usability of the library.
CA1725	<a href="#">CA1725: Parameter names should match base declaration</a>	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	<a href="#">CA1726: Use preferred terms</a>	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	<a href="#">CA1800: Do not cast unnecessarily</a>	Duplicate casts decrease performance, especially when the casts are performed in compact iteration statements.
CA1801	<a href="#">CA1801: Review unused parameters</a>	A method signature includes a parameter that is not used in the method body.
CA1802	<a href="#">CA1802: Use Literals Where Appropriate</a>	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	<a href="#">CA1804: Remove unused locals</a>	Unused local variables and unnecessary assignments increase the size of an assembly and decrease performance.
CA1806	<a href="#">CA1806: Do not ignore method results</a>	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	<a href="#">CA1809: Avoid excessive locals</a>	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	<a href="#">CA1810: Initialize reference type static fields inline</a>	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	<a href="#">CA1811: Avoid uncalled private code</a>	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	<a href="#">CA1812: Avoid uninstantiated internal classes</a>	An instance of an assembly-level type is not created by code in the assembly.
CA1813	<a href="#">CA1813: Avoid unsealed attributes</a>	The .NET Framework class library 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	<a href="#">CA1814: Prefer jagged arrays over multidimensional</a>	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.

CHECKID	WARNING	DESCRIPTION
CA1815	<a href="#">CA1815: Override equals and operator equals on value types</a>	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	<a href="#">CA1816: Call GC.SuppressFinalize correctly</a>	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	<a href="#">CA1819: Properties should not return arrays</a>	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	<a href="#">CA1820: Test for empty strings using string length</a>	Comparing strings by using the String.Length property or the String.IsNullOrEmpty method is significantly faster than using Equals.
CA1821	<a href="#">CA1821: Remove empty finalizers</a>	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	<a href="#">CA1822: Mark members as static</a>	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	<a href="#">CA1823: Avoid unused private fields</a>	Private fields were detected that do not appear to be accessed in the assembly.

CHECKID	WARNING	DESCRIPTION
CA1824	<a href="#">CA1824: Mark assemblies with NeutralResourcesLanguageAttribute</a>	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.
CA1900	<a href="#">CA1900: Value type fields should be portable</a>	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	<a href="#">CA1901: P/Invoke declarations should be portable</a>	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	<a href="#">CA1903: Use only API from targeted framework</a>	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	<a href="#">CA2000: Dispose objects before losing scope</a>	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	<a href="#">CA2001: Avoid calling problematic methods</a>	A member calls a potentially dangerous or problematic method.
CA2002	<a href="#">CA2002: Do not lock on objects with weak identity</a>	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	<a href="#">CA2003: Do not treat fibers as threads</a>	A managed thread is being treated as a Win32 thread.
CA2004	<a href="#">CA2004: Remove calls to GC.KeepAlive</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2006	<a href="#">CA2006: Use SafeHandle to encapsulate native resources</a>	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.
CA2100	<a href="#">CA2100: Review SQL queries for security vulnerabilities</a>	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	<a href="#">CA2101: Specify marshaling for P/Invoke string arguments</a>	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	<a href="#">CA2102: Catch non-CLSCopliant exceptions in general handlers</a>	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.
CA2103	<a href="#">CA2103: Review imperative security</a>	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	<a href="#">CA2104: Do not declare read only mutable reference types</a>	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	<a href="#">CA2105: Array fields should not be read only</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2106	<a href="#">CA2106: Secure asserts</a>	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	<a href="#">CA2107: Review deny and permit only usage</a>	The PermitOnly method and CodeAccessPermission.Deny security actions should be used only by those who have an advanced knowledge of .NET Framework security. Code that uses these security actions should undergo a security review.
CA2108	<a href="#">CA2108: Review declarative security on value types</a>	A public or protected value type is secured by Data Access or Link Demands.
CA2109	<a href="#">CA2109: Review visible event handlers</a>	A public or protected event-handling method was detected. Event-handling methods should not be exposed unless absolutely necessary.
CA2111	<a href="#">CA2111: Pointers should not be visible</a>	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.
CA2112	<a href="#">CA2112: Secured types should not expose fields</a>	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	<a href="#">CA2114: Method security should be a superset of type</a>	A method should not have both method-level and type-level declarative security for the same action.
CA2115	<a href="#">CA2115: Call GC.KeepAlive when using native resources</a>	This rule detects errors that might occur because an unmanaged resource is being finalized while it is still being used in unmanaged code.
CA2116	<a href="#">CA2116: APTCA methods should only call APTCA methods</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2117	<a href="#">CA2117: APTCA types should only extend APTCA base types</a>	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	<a href="#">CA2118: Review SuppressUnmanagedCodeSecurityAttribute usage</a>	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	<a href="#">CA2119: Seal methods that satisfy private interfaces</a>	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	<a href="#">CA2120: Secure serialization constructors</a>	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	<a href="#">CA2121: Static constructors should be private</a>	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	<a href="#">CA2122: Do not indirectly expose methods with link demands</a>	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	<a href="#">CA2123: Override link demands should be identical to base</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2124	<a href="#">CA2124: Wrap vulnerable finally clauses in outer try</a>	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	<a href="#">CA2126: Type link demands require inheritance demands</a>	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.
CA2127	<a href="#">CA2136: Members should not have conflicting transparency annotations</a>	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.
CA2128	<a href="#">CA2147: Transparent methods may not use security asserts</a>	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.
CA2129	<a href="#">CA2140: Transparent code must not reference security critical items</a>	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.
CA2130	<a href="#">CA2130: Security critical constants should be transparent</a>	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	<a href="#">CA2131: Security critical types may not participate in type equivalence</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2132	<a href="#">CA2132: Default constructors must be at least as critical as base type default constructors</a>	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	<a href="#">CA2133: Delegates must bind to methods with consistent transparency</a>	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	<a href="#">CA2134: Methods must keep consistent transparency when overriding base methods</a>	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	<a href="#">CA2135: Level 2 assemblies should not contain LinkDemands</a>	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.
CA2136	<a href="#">CA2136: Members should not have conflicting transparency annotations</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2137	<a href="#">CA2137: Transparent methods must contain only verifiable IL</a>	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 (MSIL). However, the rule does not contain a full IL verifier, and instead uses heuristics to catch most violations of MSIL verification.
CA2138	<a href="#">CA2138: Transparent methods must not call methods with the SuppressUnmanagedCodeSecurity attribute</a>	A security transparent method calls a method that is marked by using the SuppressUnmanagedCodeSecurityAttribute attribute.
CA2139	<a href="#">CA2139: Transparent methods may not use the HandleProcessCorruptingExceptions attribute</a>	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.
CA2140	<a href="#">CA2140: Transparent code must not reference security critical items</a>	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	<a href="#">CA2141:Transparent methods must not satisfy LinkDemands</a>	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	<a href="#">CA2142: Transparent code should not be protected with LinkDemands</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2143	<a href="#">CA2143: Transparent methods should not use security demands</a>	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	<a href="#">CA2144: Transparent code should not load assemblies from byte arrays</a>	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.
CA2145	<a href="#">CA2145: Transparent methods should not be decorated with the SuppressUnmanagedCodeSecurityAttribute attribute</a>	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	<a href="#">CA2146: Types must be at least as critical as their base types and interfaces</a>	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.
CA2147	<a href="#">CA2147: Transparent methods may not use security asserts</a>	Code that is marked as SecurityTransparentAttribute is not granted sufficient permissions to assert.
CA2149	<a href="#">CA2149: Transparent methods must not call into native code</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2151	<a href="#">CA2151: Fields with critical types should be security critical</a>	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	<a href="#">CA2200: Rethrow to preserve stack details</a>	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	<a href="#">CA2201: Do not raise reserved exception types</a>	This makes the original error difficult to detect and debug.
CA2202	<a href="#">CA2202: Do not dispose objects multiple times</a>	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	<a href="#">CA2204: Literals should be spelled correctly</a>	A literal string in a method body contains one or more words that are not recognized by the Microsoft spelling checker library.
CA2205	<a href="#">CA2205: Use managed equivalents of Win32 API</a>	An operating system invoke method is defined and a method that has the equivalent functionality is located in the .NET Framework class library.
CA2207	<a href="#">CA2207: Initialize value type static fields inline</a>	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	<a href="#">CA2208: Instantiate argument exceptions correctly</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2210	<a href="#">CA2210: Assemblies should have valid strong names</a>	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	<a href="#">CA2211: Non-constant fields should not be visible</a>	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.
CA2212	<a href="#">CA2212: Do not mark serviced components with WebMethod</a>	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	<a href="#">CA2213: Disposable fields should be disposed</a>	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	<a href="#">CA2214: Do not call overridable methods in constructors</a>	When a constructor calls a virtual method, the constructor for the instance that invokes the method may not have executed.
CA2215	<a href="#">CA2215: Dispose methods should call base class dispose</a>	If a type inherits from a disposable type, it must call the Dispose method of the base type from its own Dispose method.
CA2216	<a href="#">CA2216: Disposable types should declare finalizer</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2217	<a href="#">CA2217: Do not mark enums with FlagsAttribute</a>	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	<a href="#">CA2218: Override GetHashCode on overriding Equals</a>	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	<a href="#">CA2219: Do not raise exceptions in exception clauses</a>	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.
CA2220	<a href="#">CA2220: Finalizers should call base class finalizer</a>	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	<a href="#">CA2221: Finalizers should be protected</a>	Finalizers must use the family access modifier.
CA2222	<a href="#">CA2222: Do not decrease inherited member visibility</a>	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	<a href="#">CA2223: Members should differ by more than return type</a>	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	<a href="#">CA2224: Override equals on overloading operator equals</a>	A public type implements the equality operator but does not override Object.Equals.
CA2225	<a href="#">CA2225: Operator overloads have named alternates</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2226	<a href="#">CA2226: Operators should have symmetrical overloads</a>	A type implements the equality or inequality operator and does not implement the opposite operator.
CA2227	<a href="#">CA2227: Collection properties should be read only</a>	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	<a href="#">CA2228: Do not ship unreleased resource formats</a>	Resource files that were built by using prerelease versions of the .NET Framework might not be usable by supported versions of the .NET Framework.
CA2229	<a href="#">CA2229: Implement serialization constructors</a>	To fix a violation of this rule, implement the serialization constructor. For a sealed class, make the constructor private; otherwise, make it protected.
CA2230	<a href="#">CA2230: Use params for variable arguments</a>	A public or protected type contains a public or protected method that uses the VarArgs calling convention instead of the params keyword.
CA2231	<a href="#">CA2231: Overload operator equals on overriding ValueType.Equals</a>	A value type overrides Object.Equals but does not implement the equality operator.
CA2232	<a href="#">CA2232: Mark Windows Forms entry points with STAThread</a>	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	<a href="#">CA2233: Operations should not overflow</a>	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	<a href="#">CA2234: Pass System.Uri objects instead of strings</a>	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.

CHECKID	WARNING	DESCRIPTION
CA2235	<a href="#">CA2235: Mark all non-serializable fields</a>	An instance field of a type that is not serializable is declared in a type that is serializable.
CA2236	<a href="#">CA2236: Call base class methods on ISerializable types</a>	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	<a href="#">CA2237: Mark ISerializable types with SerializableAttribute</a>	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	<a href="#">CA2238: Implement serialization methods correctly</a>	A method that handles a serialization event does not have the correct signature, return type, or visibility.
CA2239	<a href="#">CA2239: Provide deserialization methods for optional fields</a>	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	<a href="#">CA2240: Implement ISerializable correctly</a>	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	<a href="#">CA2241: Provide correct arguments to formatting methods</a>	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	<a href="#">CA2242: Test for NaN correctly</a>	This expression tests a value against Single.NaN or Double.NaN. Use Single.IsNaN(Single) or Double.IsNaN(Double) to test the value.
CA2243	<a href="#">CA2243: Attribute string literals should parse correctly</a>	The string literal parameter of an attribute does not parse correctly for a URL, a GUID, or a version.

CHECKID	WARNING	DESCRIPTION
CA5122	<a href="#">CA5122 P/Invoke declarations should not be safe critical</a>	<p>Methods are marked as <code>SecuritySafeCritical</code> 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 <code>security safe critical</code> will not enable transparent code to call it, and is misleading for security analysis.</p>

# Cryptography Warnings

2/8/2019 • 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
<a href="#">CA5350: Do Not Use Weak Cryptographic Algorithms</a>	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.
<a href="#">CA5351 Do Not Use Broken Cryptographic Algorithms</a>	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotUseWeakCryptographicAlgorithms
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotUseBrokenCryptographicAlgorithms
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

2/8/2019 • 14 minutes to read • [Edit Online](#)

Design warnings support adherence to the .NET Framework Design Guidelines.

## In This Section

RULE	DESCRIPTION
<a href="#">CA1000: Do not declare static members on generic types</a>	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.
<a href="#">CA1001: Types that own disposable fields should be disposable</a>	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.
<a href="#">CA1002: Do not expose generic lists</a>	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.
<a href="#">CA1003: Use generic event handler instances</a>	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.
<a href="#">CA1004: Generic methods should provide type parameter</a>	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.
<a href="#">CA1005: Avoid excessive parameters on generic types</a>	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	The .NET Framework 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.
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.

Rule	Description
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.
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.

Rule	Description
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.
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.

RULE	DESCRIPTION
<a href="#">CA1060: Move P/Invokes to NativeMethods class</a>	Platform Invocation methods, such as those marked with the <a href="#">System.Runtime.InteropServices.DllImportAttribute</a> 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.
<a href="#">CA1061: Do not hide base class methods</a>	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.
<a href="#">CA1062: Validate arguments of public methods</a>	All reference arguments that are passed to externally visible methods should be checked against null.
<a href="#">CA1063: Implement IDisposable correctly</a>	All IDisposable types should implement the Dispose pattern correctly.
<a href="#">CA1064: Exceptions should be public</a>	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 <a href="#">System.Exception</a> , <a href="#">System.SystemException</a> , or <a href="#">System.ApplicationException</a> , the external code will not have sufficient information to know what to do with the exception.
<a href="#">CA1065: Do not raise exceptions in unexpected locations</a>	A method that is not expected to throw exceptions throws an exception.
<a href="#">CA2210: Assemblies should have valid strong names</a>	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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotDeclareStaticMembersOnGenericTypes
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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TypesThatOwnDisposableFieldsShouldBeDisposable
CheckId	CA1001
Category	Microsoft.Design
Breaking Change	<p>Non-breaking - If the type is not visible outside the assembly.</p> <p>Breaking - If the type is visible outside the assembly.</p>

## 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.

## 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

Do not suppress a warning from this rule.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	DoNotExposeGenericLists
Check Id	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 `System.Collections.Generic.List<T>` type, or whose signature includes a `System.Collections.Generic.List<T>` parameter.

## Rule description

`System.Collections.Generic.List<T>` is a generic collection that is designed for performance and not inheritance. `System.Collections.Generic.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 `System.Collections.Generic.List<T>`.

- `System.Collections.ObjectModel.Collection<T>`
- `System.Collections.ObjectModel.ReadOnlyCollection<T>`
- `System.Collections.ObjectModel.KeyedCollection<TKey,TItem>`

## 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 is 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

3/12/2019 • 3 minutes to read • [Edit Online](#)

Type Name	UseGenericEventHandlerInstances
Check Id	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, 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. This is no longer true in .NET. The .NET Framework introduced the [System.EventHandler<TEventArgs>](#) delegate, a generic delegate that 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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	GenericMethodsShouldProvideTypeParameter
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidExcessiveParametersOnGenericTypes
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	DoNotNestGenericTypesInMemberSignatures
Check Id	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
    End Class
End Namespace
```

```

    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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	UseGenericsWhereAppropriate
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 satisfies 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 satisfies 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

3/12/2019 • 4 minutes to read • [Edit Online](#)

Type Name	EnumsShouldHaveZeroValue
Check Id	CA1008
Category	Microsoft.Design
Breaking Change	Non-breaking - When you are prompted to add a <b>None</b> 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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DeclareEventHandlersCorrectly
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.

## 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

3/12/2019 • 3 minutes to read • [Edit Online](#)

Type Name	CollectionsShouldImplementGenericInterface
Check Id	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](#).

## 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 through static code 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](#).

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ConsiderPassingBaseTypesAsParameters
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AbstractTypesShouldNotHaveConstructors
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 through static code 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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	OverloadOperatorEqualsOnOverloadingAddAndSubtract
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkAssembliesWithClsCompliant
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:CLSCCompliant(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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkAssembliesWithAssemblyVersion
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).

The .NET Framework 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. See the following example.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkAssembliesWithComVisible
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	MarkAttributesWithAttributeUsage
Check Id	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 `A11` 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DefineAccessorsForAttributeArguments
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidNamespacesWithFewTypes
CheckId	CA1020
Category	Microsoft.Design
Breaking Change	Breaking

## Cause

A namespace other than the global namespace contains fewer than five types.

## 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

2/8/2019 • 6 minutes to read • [Edit Online](#)

TypeName	AvoidOutParameters
CheckId	CA1021
Category	Microsoft.Design
Breaking Change	Breaking

## Cause

A public or protected method in a public type has an `out` parameter.

## 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.

## 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;
                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 reassigned 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	IndexersShouldNotBeMultidimensional
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

3/12/2019 • 4 minutes to read • [Edit Online](#)

Type Name	UsePropertiesWhereAppropriate
Check Id	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 perceptibly 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 through static code 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
{
    public class 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);
        }
    }
}

```

```
    .CustomerName, .CustomerID, .When);
}

// 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReplaceRepetitiveArgumentsWithParamsArray
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DefaultParametersShouldNotBeUsed
CheckId	CA1026
Category	Microsoft.Design
Breaking Change	Breaking

## Cause

An externally visible type contains an externally visible method that uses a default parameter.

## 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	MarkEnumsWithFlags
Check Id	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 through static code 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

3/12/2019 • 3 minutes to read • [Edit Online](#)

TypeName	EnumStorageShouldBeInt32
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 through static code 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 UIInteger
        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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	UseEventsWhereAppropriate
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 Framework 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, you should 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 through static code 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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	DoNotCatchGeneralExceptionTypes
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.

## 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.

## 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);  
        }  
  
        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)
                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);
            }

            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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	ImplementStandardExceptionConstructors
Check Id	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 static code analysis as opposed to [Roslyn-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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	InterfaceMethodsShouldBeCallableByChildTypes
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	NestedTypesShouldNotBeVisible
CheckId	CA1034
Category	Microsoft.Design
Breaking Change	Breaking

## Cause

An externally visible type contains an externally visible type declaration. Nested enumerations and protected types 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ICollectionImplementationsHaveStronglyTypedMembers
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.

## 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](#)

[CA1039: Lists are strongly typed](#)

## See also

- [System.Array](#)
- [System.Collections.IEnumerable](#)
- [System.Collections.ICollection](#)
- [System.Object](#)

# CA1036: Override methods on comparable types

3/12/2019 • 4 minutes to read • [Edit Online](#)

TypeName	OverrideMethodsOnComparableTypes
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 through static code 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.OrdinalIgnoreCase) > 0 || string.Compare(v, "A", StringComparison.OrdinalIgnoreCase) < 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)
```

```

        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 string1 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	EnumeratorsShouldBeStronglyTyped
Check Id	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](#)

## 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.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.
    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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	ListsAreStronglyTyped
Check Id	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`

## 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidEmptyInterfaces
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 through static code 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ProvideObsoleteAttributeMessage
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 through static code 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	UseIntegralOrStringArgumentForIndexers
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 through static code 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	PropertiesShouldNotBeWriteOnly
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 through static code 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

2/8/2019 • 6 minutes to read • [Edit Online](#)

TypeName	DoNotPassTypesByReference
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.

## 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.

```

        }

        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 reassigned 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotOverloadOperatorEqualsOnReferenceTypes
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotDeclareProtectedMembersInSealedTypes
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotDeclareVirtualMembersInSealedTypes
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TypesThatOwnNativeResourcesShouldBeDisposable
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 Overrides 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DeclareTypesInNamespaces
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	DoNotDeclareVisibleInstanceFields
Check Id	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 is 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; very little performance gain is associated with the use of externally visible fields over properties.

Externally visible refers to `public`, `protected`, and `protected internal` (`Public`, `Protected`, and `Protected Friend` in Visual Basic) accessibility levels.

## 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

Do not suppress a warning from this rule. Externally visible fields do not provide any benefits that are unavailable to properties. Additionally, public fields cannot be protected by [Link Demands](#). See [CA2112: Secured types should not expose fields](#).

## Configurability

If you're running this rule from [FxCop analyzers](#) (and not through static code 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 sealed

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	StaticHolderTypesShouldBeSealed
Check Id	CA1052
Category	Microsoft.Design
Breaking Change	Breaking

## Cause

A non-abstract type contains only static members and is not declared with the [sealed \(NotInheritable\)](#) 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 `sealed` or `NotInheritable` modifier to prohibit its use as a base type. This rule does not fire for abstract classes.

## How to fix violations

To fix a violation of this rule, mark the type as `sealed` or `NotInheritable`. If you're targeting .NET Framework 2.0 or later, a better approach is to mark the type as `static` or `Shared`. In this manner, you don't have to declare a private constructor to prevent the class from being created.

## When to suppress warnings

Suppress a warning from this rule only if the type is designed to be inherited. The absence of the `sealed` or `NotInheritable` modifier suggests that the type is useful as a base type.

## Configurability

If you're running this rule from [FxCop analyzers](#) (and not through static code 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:

```
using System;

namespace DesignLibrary
{
    public class StaticMembers
    {
        static int someField;

        public static int SomeProperty
        {
            get
            {
                return someField;
            }
            set
            {
                someField = value;
            }
        }

        StaticMembers() {}

        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#:

```

using System;

namespace DesignLibrary
{
    public static class StaticMembers
    {
        private static int someField;

        public static int SomeProperty
        {
            get { return someField; }
            set { someField = value; }
        }

        public static void SomeMethod()
        {
        }

        public static event SomeDelegate SomeEvent;
    }

    public delegate void SomeDelegate();
}

```

## Related rules

- [CA1053: Static holder types should not have constructors](#)

# CA1053: Static holder types should not have constructors

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	StaticHolderTypesShouldNotHaveConstructors
CheckId	CA1053
Category	Microsoft.Design
Breaking Change	Breaking

## Cause

A public or nested public type declares only static members and has a public or protected default constructor.

## Rule description

The 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 or make it private.

### NOTE

Some compilers automatically create a public default constructor if the type does not define any constructors. If this is the case with your type, add a private default constructor to eliminate the violation.

## When to suppress warnings

Do not suppress a warning from this rule. The presence of the constructor suggests that the type is not a static type.

## Example

The following example shows a type that violates this rule. Notice that there is no default constructor in the source code. When this code is compiled into an assembly, the C# compiler will insert a default constructor, which will violate this rule. To correct this, declare a private constructor.

```
using System;

namespace DesignLibrary
{
    public class NoInstancesNeeded
    {
        // Violates rule: StaticHolderTypesShouldNotHaveConstructors.
        // Uncomment the following line to correct the violation.
        // private NoInstancesNeeded() {}

        public static void Method1() {}
        public static void Method2() {}
    }
}
```

# CA1054: URI parameters should not be strings

3/12/2019 • 3 minutes to read • [Edit Online](#)

Type Name	UriParametersShouldNotBeStrings
Check Id	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 through static code 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

3/12/2019 • 3 minutes to read • [Edit Online](#)

Type Name	UriReturnValuesShouldNotBeStrings
Check Id	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 through static code 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

3/12/2019 • 3 minutes to read • [Edit Online](#)

Type Name	UriPropertiesShouldNotBeStrings
Check Id	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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	StringUriOverloadsCallSystemUriOverloads
Check Id	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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TypesShouldNotExtendCertainBaseTypes
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

For .NET Framework version 1, it was recommended to derive new exceptions from [ApplicationException](#). The recommendation has changed and new 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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MembersShouldNotExposeCertainConcreteTypes
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](#).

## 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
<a href="#">XPathDocument</a>	<a href="#">System.Xml.XPath.IXPathNavigable</a> .  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

2/8/2019 • 6 minutes to read • [Edit Online](#)

TypeName	MovePInvokesToNativeMethodsClass
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotHideBaseClassMethods
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ValidateArgumentsOfPublicMethods
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).

## 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.

## 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("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("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,
              PassThroughNonNull(other).Age)
    {
    }

    // Null check method
    private static Person PassThroughNonNull(Person person)
    {
        if (person == null)
            throw new ArgumentNullException("person");
        return person;
    }
}
```

# CA1063: Implement IDisposable correctly

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	ImplementIDisposableCorrectly
Check Id	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 reoverridden.
- 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 through static code 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 IntPtr nativeResource = Marshal.AllocHGlobal(100);
    private AnotherResource managedResource = new AnotherResource();

    // Dispose() calls Dispose(true)
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // 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);
    }

    // The bulk of the clean-up code is implemented in Dispose(bool)
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // free managed resources
            if (managedResource != null)
            {
                managedResource.Dispose();
                managedResource = null;
            }
        }
        // free native resources if there are any.
        if (nativeResource != IntPtr.Zero)
        {
            Marshal.FreeHGlobal(nativeResource);
            nativeResource = IntPtr.Zero;
        }
    }
}
```

## See also

- [Dispose pattern \(framework design guidelines\)](#)
- [Clean up unmanaged resources \(.NET guide\)](#)

# CA1064: Exceptions should be public

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ExceptionsShouldBePublic
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotRaiseExceptionsInUnexpectedLocations
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](#)

# CA2210: Assemblies should have valid strong names

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AssembliesShouldHaveValidStrongNames
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.

## 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) provided by the .NET Framework SDK.

- For the .NET Framework v1.0 or v1.1, use either the [System.Reflection.AssemblyKeyFileAttribute](#) or [System.Reflection.AssemblyKeyNameAttribute](#) attribute.
- 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++).

### 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) that is provided by the .NET Framework SDK. For more information, see [Sn.exe \(Strong Name Tool\)](#).

## 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 \(Strong Name Tool\)](#)

# Globalization Warnings

2/8/2019 • 2 minutes to read • [Edit Online](#)

Globalization warnings support world-ready libraries and applications.

## In This Section

RULE	DESCRIPTION
<a href="#">CA1300: Specify MessageBoxOptions</a>	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.
<a href="#">CA1301: Avoid duplicate accelerators</a>	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.
<a href="#">CA1302: Do not hardcode locale specific strings</a>	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.
<a href="#">CA1303: Do not pass literals as localized parameters</a>	An externally visible method passes a string literal as a parameter to a constructor or method in the .NET Framework class library, and that string should be localizable.
<a href="#">CA1304: Specify CultureInfo</a>	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.
<a href="#">CA1305: Specify IFormatProvider</a>	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.
<a href="#">CA1306: Set locale for data types</a>	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
<a href="#">CA1307: Specify StringComparison</a>	A string comparison operation uses a method overload that does not set a StringComparison parameter.
<a href="#">CA1308: Normalize strings to uppercase</a>	Strings should be normalized to uppercase. A small group of characters cannot make a round trip when they are converted to lowercase.
<a href="#">CA1309: Use ordinal StringComparison</a>	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.
<a href="#">CA2101: Specify marshaling for P/Invoke string arguments</a>	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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	SpecifyMessageBoxOptions
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.
    End Sub
End Class
```

```

'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.CurrentCulture.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()

```

```

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.CurrentCulture.TextInfo.IsRightToLeft;
    }
}
}

```

## See also

- [System.Resources.ResourceManager](#)
- [Resources in desktop apps \(.NET Framework\)](#)

# CA1301: Avoid duplicate accelerators

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	AvoidDuplicateAccelerators
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotHardcodeLocaleSpecificStrings
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotPassLiteralsAsLocalizedParameters
CheckId	CA1303
Category	Microsoft.Globalization
Breaking Change	Non Breaking

## Cause

A method passes a string literal as a parameter to a constructor or method in the .NET Framework class library 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 parameter or property name contains "Text", "Message", or "Caption".
- The name of the string parameter that is passed to a `Console.WriteLine` method is either "value" or "format".

## 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 named, or by marking these items as conditional.

## 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.CurrentCulture))
            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.CurrentCulture));
            }
        }
    }
}
```

## See also

[Resources in Desktop Apps](#)

# CA1304: Specify CultureInfo

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	SpecifyCultureInfo
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](#)

## 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 Framework 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.CurrentCulture](#) 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.

## 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	SpecifyIFormatProvider
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 Framework methods that are documented as ignoring the [IFormatProvider](#) parameter. The rule also ignores the following methods:

- [CreateInstance](#)
- [ResourceManager.GetObject](#)
- [ResourceManager.GetString](#)

## 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 Framework 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	SetLocaleForDataTypes
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.CurrentCulture](#)
- [System.Globalization.CultureInfo.InvariantCulture](#)

# CA1307: Specify StringComparison

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	SpecifyStringComparison
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	NormalizeStringsToUppercase
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	UseOrdinalStringComparison
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	SpecifyMarshalingForPInvokeStringArguments
Check Id	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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Interoperability warnings support interaction with COM clients.

## In This Section

RULE	DESCRIPTION
<a href="#">CA1400: P/Invoke entry points should exist</a>	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.
<a href="#">CA1401: P/Invokes should not be visible</a>	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.
<a href="#">CA1402: Avoid overloads in COM visible interfaces</a>	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.
<a href="#">CA1403: Auto layout types should not be COM visible</a>	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 the .NET Framework, which will break COM clients that expect a specific layout.
<a href="#">CA1404: Call GetLastError immediately after P/Invoke</a>	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.
<a href="#">CA1405: COM visible type base types should be COM visible</a>	A COM-visible type derives from a type that is not COM-visible.
<a href="#">CA1406: Avoid Int64 arguments for Visual Basic 6 clients</a>	Visual Basic 6 COM clients cannot access 64-bit integers.
<a href="#">CA1407: Avoid static members in COM visible types</a>	COM does not support static methods.
<a href="#">CA1408: Do not use AutoDual ClassInterfaceType</a>	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 <a href="#">System.Runtime.InteropServices.ComRegisterFunctionAttribute</a> attribute but does not declare a method that is marked by using the <a href="#">System.Runtime.InteropServices.ComUnregisterFunctionAttribute</a> attribute, or vice versa.
CA1411: COM registration methods should not be visible	A method that is marked by using the <a href="#">System.Runtime.InteropServices.ComRegisterFunctionAttribute</a> attribute or the <a href="#">System.Runtime.InteropServices.ComUnregisterFunctionAttribute</a> attribute is externally visible.
CA1412: Mark ComSource Interfaces as IDispatch	A type is marked by using the <a href="#">System.Runtime.InteropServices.ComSourceInterfacesAttribute</a> attribute, and at least one of the specified interfaces is not marked by using the <a href="#">System.Runtime.InteropServices.InterfaceTypeAttribute</a> attribute set to <code>ComInterfaceType.InterfaceIsIDispatch</code> .
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 <a href="#">System.Threading.NativeOverlapped</a> structure.

# CA1400: P/Invoke entry points should exist

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	PInvokeEntryPointsShouldExist
Check Id	CA1400
Category	Microsoft.Interopability
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	PInvokesShouldNotBeVisible
CheckId	CA1401
Category	Microsoft.Interopability
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidOverloadsInComVisibleInterfaces
CheckId	CA1402
Category	Microsoft.Interopability
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AutoLayoutTypesShouldNotBeComVisible
CheckId	CA1403
Category	Microsoft.Interopability
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 the .NET Framework, 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	CallGetLastErrorImmediatelyAfterPInvoke
Check Id	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.IsValidId](#)

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ComVisibleTypeBaseTypesShouldBeComVisible
CheckId	CA1405
Category	Microsoft.Interopability
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidInt64ArgumentsForVB6Clients
CheckId	CA1406
Category	Microsoft.Interopability
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	AvoidStaticMembersInComVisibleTypes
Check Id	CA1407
Category	Microsoft.Interopability
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotUseAutoDualClassInterfaceType
CheckId	CA1408
Category	Microsoft.Interopability
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ComVisibleTypesShouldBeCreatable
CheckId	CA1409
Category	Microsoft.Interopability
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ComRegistrationMethodsShouldBeMatched
CheckId	CA1410
Category	Microsoft.Interopability
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 Framework 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](#)
- [Registering Assemblies with COM](#)
- [Regasm.exe \(Assembly Registration Tool\)](#)

# CA1411: COM registration methods should not be visible

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ComRegistrationMethodsShouldNotBeVisible
CheckId	CA1411
Category	Microsoft.Interopability
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkComSourceInterfacesAsIDispatch
CheckId	CA1412
Category	Microsoft.Interopability
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 `InterfaceIsDispatch` 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 `InterfaceIsDispatch` 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

CA1408: Do not use AutoDual ClassInterfaceType

## See also

- [Interoperating with Unmanaged Code](#)

# CA1413: Avoid non-public fields in COM visible value types

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidNonpublicFieldsInComVisibleValueTypes
CheckId	CA1413
Category	Microsoft.Interopability
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkBooleanPInvokeArgumentsWithMarshalAs
CheckId	CA1414
Category	Microsoft.Interopability
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DeclarePInvokesCorrectly
CheckId	CA1415
Category	Microsoft.Interopability
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](#)

# Maintainability Warnings

2/8/2019 • 2 minutes to read • [Edit Online](#)

Maintainability warnings support library and application maintenance.

## In This Section

RULE	DESCRIPTION
<a href="#">CA1500: Variable names should not match field names</a>	An instance method declares a parameter or a local variable whose name matches an instance field of the declaring type, which leads to errors.
<a href="#">CA1501: Avoid excessive inheritance</a>	A type is more than four levels deep in its inheritance hierarchy. Deeply nested type hierarchies can be difficult to follow, understand, and maintain.
<a href="#">CA1502: Avoid excessive complexity</a>	This rule measures the number of linearly independent paths through the method, which is determined by the number and complexity of conditional branches.
<a href="#">CA1504: Review misleading field names</a>	The name of an instance field starts with "s_ ", or the name of a static (Shared in Visual Basic) field starts with "m_ ".
<a href="#">CA1505: Avoid unmaintainable code</a>	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.
<a href="#">CA1506: Avoid excessive class coupling</a>	This rule measures class coupling by counting the number of unique type references that a type or method contains.

## See Also

- [Measuring Complexity and Maintainability of Managed Code](#)

# CA1500: Variable names should not match field names

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	VariableNamesShouldNotMatchFieldNames
CheckId	CA1500
Category	Microsoft.Maintainability
Breaking Change	<p>When fired on a parameter that has the same name as a field:</p> <ul style="list-style-type: none"><li>- Non-breaking - If both the field and method that declares the parameter cannot be seen outside the assembly, regardless of the change you make.</li><li>- Breaking - If you change the name of the field and can be seen outside the assembly.</li><li>- Breaking - If you change the name of the parameter and the method that declares it can be seen outside the assembly.</li></ul> <p>When fired on a local variable that has the same name as a field:</p> <ul style="list-style-type: none"><li>- Non-breaking - If the field cannot be seen outside the assembly, regardless of the change you make.</li><li>- Non-breaking - If you change the name of the local variable and do not change the name of the field.</li><li>- Breaking - If you change the name of the field and it can be seen outside the assembly.</li></ul>

## 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

4/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidExcessiveInheritance
CheckId	CA1501
Category	Microsoft.Maintainability
Breaking Change	Breaking

## Cause

A type is more than four levels deep in its inheritance hierarchy.

## 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.

## 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

4/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidExcessiveComplexity
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 runtime 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewMisleadingFieldNames
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\_".

## 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

4/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidUnmantainableCode
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

4/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidExcessiveClassCoupling
CheckId	CA1506
Category	Microsoft.Maintainability
Breaking Change	Breaking

## Cause

A type or method is coupled with many other types.

## 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](#)

# Mobility Warnings

2/8/2019 • 2 minutes to read • [Edit Online](#)

Mobility warnings support efficient power usage.

## In This Section

RULE	DESCRIPTION
<a href="#">CA1600: Do not use idle process priority</a>	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.
<a href="#">CA1601: Do not use timers that prevent power state changes</a>	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotUseIdleProcessPriority
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotUseTimersThatPreventPowerStateChanges
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

2/8/2019 • 4 minutes to read • [Edit Online](#)

Naming warnings support adherence to the naming conventions of the .NET Framework Design Guidelines.

## In This Section

RULE	DESCRIPTION
<a href="#">CA1700: Do not name enum values 'Reserved'</a>	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.
<a href="#">CA1713: Events should not have before or after prefix</a>	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.
<a href="#">CA1714: Flags enums should have plural names</a>	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.
<a href="#">CA1704: Identifiers should be spelled correctly</a>	The name of an externally visible identifier contains one or more words that are not recognized by the Microsoft spelling checker library.
<a href="#">CA1708: Identifiers should differ by more than case</a>	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.
<a href="#">CA1715: Identifiers should have correct prefix</a>	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".
<a href="#">CA1720: Identifiers should not contain type names</a>	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.
<a href="#">CA1722: Identifiers should not have incorrect prefix</a>	By convention, only certain programming elements have names that begin with a specific prefix.
<a href="#">CA1711: Identifiers should not have incorrect suffix</a>	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 namespaces that are defined in the .NET Framework class library. 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'

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	DoNotNameEnumValuesReserved
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ResourceStringCompoundWordsShouldBeCasedCorrectly
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	CompoundWordsShouldBeCasedCorrectly
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ResourceStringsShouldBeSpelledCorrectly
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	IdentifiersShouldBeSpelledCorrectly
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	IdentifiersShouldNotContainUnderscores
Check Id	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 a warning from this rule.

## Related rules

- [CA1709: Identifiers should be cased correctly](#)
- [CA1708: Identifiers should differ by more than case](#)

# CA1708: Identifiers should differ by more than case

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	IdentifiersShouldDifferByMoreThanCase
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 the .NET Framework.

## Configurability

If you're running this rule from [FxCop analyzers](#) (and not through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	IdentifiersShouldBeCasedCorrectly
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	IdentifiersShouldHaveCorrectSuffix
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.Queue	Collection or Queue
System.Collections.Stack	Collection or Stack
System.Collections.Generic.ICollection<T>	Collection

BASE TYPE/INTERFACE	SUFFIX
<code>System.Collections.Generic.IDictionary&lt; TKey, TValue &gt;</code>	Dictionary
<code>System.Data.DataSet</code>	DataSet
<code>System.Data.DataTable</code>	Collection or DataTable
<code>System.IO.Stream</code>	Stream
<code>System.Security.IPermission</code>	Permission
<code>System.Security.Policy.IMembershipCondition</code>	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](#) 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 through static code 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](#).

## Related rules

[CA1711: Identifiers should not have incorrect suffix](#)

## See also

- [Attributes](#)
- [Handling and raising events](#)

# CA1711: Identifiers should not have incorrect suffix

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	IdentifiersShouldNotHaveIncorrectSuffix
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	<a href="#">System.Attribute</a>
Collection	<a href="#">System.Collections.ICollection</a> <a href="#">System.Collections.IEnumerable</a> <a href="#">System.Collections.Queue</a> <a href="#">System.Collections.Stack</a> <a href="#">System.Collections.Generic.ICollection&lt;T&gt;</a> <a href="#">System.Data.DataSet</a> <a href="#">System.Data.DataTable</a>
Dictionary	<a href="#">System.Collections.IDictionary</a> <a href="#">System.Collections.Generic.IDictionary&lt; TKey, TValue &gt;</a>
EventArgs	<a href="#">System.EventArgs</a>
EventHandler	An event-handler delegate

SUFFIX	BASE TYPE/INTERFACE
Exception	System.Exception
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 through static code 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.ca1711.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

- [CA1710: Identifiers should have correct suffix](#)

## See also

- [Attributes](#)

- Handling and raising events

# CA1712: Do not prefix enum values with type name

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotPrefixEnumValuesWithTypeName
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](#)

## See also

- [System.Enum](#)

# CA1713: Events should not have before or after prefix

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	EventsShouldNotHaveBeforeOrAfterPrefix
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	FlagsEnumsShouldHavePluralNames
Check Id	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 through static code 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

3/12/2019 • 3 minutes to read • [Edit Online](#)

TypeName	IdentifiersShouldHaveCorrectPrefix
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 through static code 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	IdentifiersShouldNotMatchKeywords
Check Id	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 this is [configurable](#).

## 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

If you're running this rule from [FxCop analyzers](#) (and not through static code 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](#).

# CA1717: Only FlagsAttribute enums should have plural names

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	OnlyFlagsEnumsShouldHavePluralNames
Check Id	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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ParameterNamesShouldNotMatchMemberNames
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	IdentifiersShouldNotContainTypeNames
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`
- `Obj`
- `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 through static code 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	PropertyNameShouldNotMatchGetMethods
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.

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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	IdentifiersShouldNotHaveIncorrectPrefix
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	ParameterNamesShouldMatchBaseDeclaration
Check Id	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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TypeNamesShouldNotMatchNamespaces
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.

# CA1726: Use preferred terms

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	UsePreferredTerms
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

OBSOLETE TERM	PREFERRED TERM
Hasnt	HasNot
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

2/8/2019 • 3 minutes to read • [Edit Online](#)

Performance warnings support high-performance libraries and applications.

## In This Section

RULE	DESCRIPTION
<a href="#">CA1800: Do not cast unnecessarily</a>	Duplicate casts decrease performance, especially when the casts are performed in compact iteration statements.
<a href="#">CA1801: Review unused parameters</a>	A method signature includes a parameter that is not used in the method body.
<a href="#">CA1802: Use Literals Where Appropriate</a>	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.
<a href="#">CA1804: Remove unused locals</a>	Unused local variables and unnecessary assignments increase the size of an assembly and decrease performance.
<a href="#">CA1806: Do not ignore method results</a>	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.
<a href="#">CA1809: Avoid excessive locals</a>	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.
<a href="#">CA1810: Initialize reference type static fields inline</a>	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.
<a href="#">CA1811: Avoid uncalled private code</a>	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.
<a href="#">CA1812: Avoid uninstantiated internal classes</a>	An instance of an assembly-level type is not created by code in the assembly.

Rule	Description
CA1813: Avoid unsealed attributes	The .NET Framework class library 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.

# CA1800: Do not cast unnecessarily

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	DoNotCastUnnecessarily
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.

## 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 \(C# reference\)](#)

# CA1802: Use Literals Where Appropriate

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	UseLiteralsWhereAppropriate
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 fields, but this is [configurable](#).

## Rule description

The value of a `static readonly` field is computed at runtime 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 runtime 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 runtime.

## How to fix violations

To fix a violation of this rule, replace the `static` and `readonly` modifiers with the `const` modifier.

## 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 through static code 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.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](#).

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	RemoveUnusedLocals
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 that the C# compiler that is included with .NET Framework 2.0 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](#)

# CA1809: Avoid excessive locals

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidExcessiveLocals
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	InitializeReferenceTypeStaticFieldsInline
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidUncalledPrivateCode
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.

## Related rules

[CA1812: Avoid uninstantiated internal classes](#)

[CA1801: Review unused parameters](#)

[CA1804: Remove unused locals](#)

# CA1812: Avoid uninstantiated internal classes

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidUninstantiatedInternalClasses
CheckId	CA1812
Category	Microsoft.Performance
Breaking Change	Non-breaking

## Cause

An instance of an assembly-level type is not created by code in the assembly.

## 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 define `static` (`Shared` in Visual Basic) methods only.

If you apply `System.Runtime.CompilerServices.InternalsVisibleToAttribute` to the assembly that is being analyzed, this rule will not occur on any constructors that are marked as `internal` because you cannot tell whether a field is being used by another `friend` assembly.

Even though you cannot work around this limitation in Visual Studio Code Analysis, the external stand-alone FxCop will occur on internal constructors if every `friend` assembly is present in the analysis.

## How to fix violations

To fix a violation of this rule, remove the type or add the 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:

- A private constructor for types that target .NET Framework versions 1.0 and 1.1.
- The `static` (`Shared` in Visual Basic) modifier for types that target .NET Framework 2.0.

## 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. For example, classes that implement [System.Configuration.IConfigurationSectionHandler](#) or [System.Web.IHttpHandler](#).
- The class is passed as a generic type parameter that has a new constraint. For example, the following example will raise this rule.

```
internal class MyClass
{
    public DoSomething()
    {
    }
}
public class MyGeneric<T> where T : new()
{
    public T Create()
    {
        return new T();
    }
}
// [...]
MyGeneric<MyClass> mc = new MyGeneric<MyClass>();
mc.Create();
```

In these situations, we recommended you suppress this warning.

## Related rules

[CA1811: Avoid uncalled private code](#)

[CA1801: Review unused parameters](#)

[CA1804: Remove unused locals](#)

# CA1813: Avoid unsealed attributes

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidUnsealedAttributes
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

The .NET Framework class library 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	PreferJaggedArraysOverMultidimensional
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.

```
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	OverrideEqualsAndOperatorEqualsOnValueTypes
Check Id	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 through static code 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](#)

# CA1819: Properties should not return arrays

3/12/2019 • 4 minutes to read • [Edit Online](#)

Type Name	PropertiesShouldNotReturnArrays
Check Id	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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	TestForEmptyStringsUsingStringLength
Check Id	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 [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 [Length](#) property and test for the null string. If targeting .NET Framework 2.0 or later, 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	RemoveEmptyFinalizers
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 is involved in tracking object lifetime. The garbage collector will run the finalizer before it collects the object. This means that two collections will be 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
```

## When to suppress warnings

Do not suppress a message from this rule. Failure to suppress finalization decreases performance and provides no benefits.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkMembersAsStatic
CheckId	CA1822
Category	Microsoft.Performance
Breaking Change	<p>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 <code>this</code> keyword.</p> <p>Breaking - If you change the member from an instance member to a static member and it is visible outside the assembly.</p>

## 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 runtime 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidUnusedPrivateFields
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	MarkAssembliesWithNeutralResourcesLanguage
Check Id	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](#)

# Portability Warnings

2/8/2019 • 2 minutes to read • [Edit Online](#)

Portability warnings support portability across different operating systems.

## In This Section

RULE	DESCRIPTION
<a href="#">CA1900: Value type fields should be portable</a>	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.
<a href="#">CA1901: P/Invoke declarations should be portable</a>	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.
<a href="#">CA1903: Use only API from targeted framework</a>	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ValueTypeFieldsShouldBePortable
CheckId	CA1900
Category	Microsoft.Portability
Breaking Change	<p>Breaking - If the field can be seen outside the assembly.</p> <p>Non-breaking - If the field is not visible outside the assembly.</p>

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	PInvokeDeclarationsShouldBePortable
Check Id	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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	UseOnlyApiFromTargetedFramework
CheckId	CA1903
Category	Microsoft.Portability
Breaking Change	<p>Breaking - when fired against the signature of an externally visible member or type.</p> <p>Non-Breaking - when fired in the body of a method.</p>

## 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.

## 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

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 [Targeting a Specific .NET Framework Version](#).

## 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](#)
- [Targeting a Specific .NET Framework Version](#)

# Reliability Warnings

2/8/2019 • 2 minutes to read • [Edit Online](#)

Reliability warnings support library and application reliability, such as correct memory and thread usage.

## In This Section

RULE	DESCRIPTION
<a href="#">CA2000: Dispose objects before losing scope</a>	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.
<a href="#">CA2001: Avoid calling problematic methods</a>	A member calls a potentially dangerous or problematic method.
<a href="#">CA2002: Do not lock on objects with weak identity</a>	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.
<a href="#">CA2003: Do not treat fibers as threads</a>	A managed thread is being treated as a Win32 thread.
<a href="#">CA2004: Remove calls to GC.KeepAlive</a>	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.
<a href="#">CA2006: Use SafeHandle to encapsulate native resources</a>	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.

# CA2000: Dispose objects before losing scope

4/16/2019 • 4 minutes to read • [Edit Online](#)

TypeName	DisposeObjectsBeforeLosingScope
CheckId	CA2000
Category	Microsoft.Reliability
Breaking Change	Non-breaking

## Cause

A local object of a [IDisposable](#) type is created but the object is not disposed before all references to the object are out of scope.

## 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.

## 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.

Note that you can use the `using` statement (`Using` in Visual Basic) to wrap objects that implement [IDisposable](#). Objects that are wrapped in this manner will automatically be disposed at the close of the `using` block.

The following are some situations where the `using` statement is not enough to protect [IDisposable](#) objects and can cause CA2000 to occur.

- Returning a disposable object requires that the object is constructed in a try/finally block outside a `using` block.
- Initializing members of a disposable object should not be done in the constructor of a `using` statement.
- Nesting constructors that are protected only by one exception handler. For example,

```
using (StreamReader sr = new StreamReader(new FileStream("C:\myfile.txt", FileMode.Create)))
{ ... }
```

causes CA2000 to occur because a failure in the construction of the `StreamReader` object can result in the `FileStream` object never being closed.

- 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 have called a method on your object that calls `Dispose`, such as `Close`, or if the method that raised the warning returns an `IDisposable` object wraps your object.

## Related rules

[CA2213: Disposable fields should be disposed](#)

[CA2202: Do not dispose objects multiple times](#)

## Example

If you are 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidCallingProblematicMethods
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
<a href="#">System.GC.Collect</a>	Calling GC.Collect can significantly affect application performance and is rarely necessary. For more information, see <a href="#">Rico Mariani's Performance Tidbits</a> blog entry on MSDN.
<a href="#">System.Threading.Thread.Resume</a> <a href="#">System.Threading.Thread.Suspend</a>	Thread.Suspend and Thread.Resume have been deprecated because of their unpredictable behavior. Use other classes in the <a href="#">System.Threading</a> namespace, such as <a href="#">Monitor</a> , <a href="#">Mutex</a> , and <a href="#">Semaphore</a> , to synchronize threads or protect resources.
<a href="#">System.Runtime.InteropServices.SafeHandle.DangerousGetHandle</a>	The DangerousGetHandle method poses a security risk because it can return a handle that is not valid. See the <a href="#">DangerousAddRef</a> and the <a href="#">DangerousRelease</a> methods for more information about how to use the DangerousGetHandle method safely.
<a href="#">System.Reflection.Assembly.LoadFrom</a> <a href="#">System.Reflection.Assembly.LoadFile</a> <a href="#">System.Reflection.Assembly.LoadWithPartialName</a>	These methods can load assemblies from unexpected locations. For example, see Suzanne Cook's .NET CLR Notes blog posts <a href="#">LoadFile vs. LoadFrom</a> and <a href="#">Choosing a Binding Context</a> for information about methods that load assemblies.

METHOD	DESCRIPTION
<a href="#">CoSetProxyBlanket</a> (Ole32)	By the time the user code starts executing in a managed process, it is too late to reliably call CoSetProxyBlanket. The common language runtime (CLR) takes initialization actions that may prevent the users P/Invoke from succeeding.
<a href="#">CoInitializeSecurity</a> (Ole32)	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

You should 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotLockOnObjectsWithWeakIdentity
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](#)

## 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

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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotTreatFibersAsThreads
CheckId	CA2003
Category	Microsoft.Reliability
Breaking Change	Non-breaking

## Cause

A managed thread is being treated as a Win32 thread.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	RemoveCallsToGCKeepAlive
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

3/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	UseSafeHandleToEncapsulateNativeResources
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotDirectlyAwaitATaskAnalyzer
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 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.

This rule was introduced with [FxCop analyzers](#) and doesn't exist in "legacy" (static code analysis) FxCop.

## 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

- [Should I await a task with ConfigureAwait\(false\)?](#)
- [Install FxCop analyzers in Visual Studio](#)

# Security warnings

3/19/2019 • 11 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
<a href="#">CA2100: Review SQL queries for security vulnerabilities</a>	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.
<a href="#">CA2102: Catch non-CLSCompliant exceptions in general handlers</a>	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.
<a href="#">CA2103: Review imperative security</a>	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.
<a href="#">CA2104: Do not declare read only mutable reference types</a>	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.
<a href="#">CA2105: Array fields should not be read only</a>	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.
<a href="#">CA2106: Secure asserts</a>	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.
<a href="#">CA2107: Review deny and permit only usage</a>	Using the PermitOnly method and CodeAccessPermission.Deny security actions should be used only by those with an advanced knowledge of .NET Framework security. Code that uses these security actions should undergo a security review.
<a href="#">CA2108: Review declarative security on value types</a>	A public or protected value type is secured by Data Access or Link Demands.

Rule	Description
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.
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.

Rule	Description
<a href="#">CA2122: Do not indirectly expose methods with link demands</a>	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.
<a href="#">CA2123: Override link demands should be identical to base</a>	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.
<a href="#">CA2124: Wrap vulnerable finally clauses in outer try</a>	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.
<a href="#">CA2126: Type link demands require inheritance demands</a>	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.
<a href="#">CA2130: Security critical constants should be transparent</a>	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.
<a href="#">CA2131: Security critical types may not participate in type equivalence</a>	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.
<a href="#">CA2132: Default constructors must be at least as critical as base type default constructors</a>	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.
<a href="#">CA2133: Delegates must bind to methods with consistent transparency</a>	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.

Rule	Description
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.
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 <a href="#">AccessViolationException</a> . 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.

Rule	Description
<a href="#">CA2142: Transparent code should not be protected with LinkDemands</a>	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.
<a href="#">CA2143: Transparent methods should not use security demands</a>	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.
<a href="#">CA2144: Transparent code should not load assemblies from byte arrays</a>	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.
<a href="#">CA2145: Transparent methods should not be decorated with the SuppressUnmanagedCodeSecurityAttribute</a>	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.
<a href="#">CA2146: Types must be at least as critical as their base types and interfaces</a>	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.
<a href="#">CA2147: Transparent methods may not use security asserts</a>	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.
<a href="#">CA2149: Transparent methods must not call into native code</a>	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.
<a href="#">CA2151: Fields with critical types should be security critical</a>	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.

RULE	DESCRIPTION
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.
CA2153: Avoid Handling Corrupted State Exceptions	<a href="#">Corrupted State Exceptions (CSE)</a> 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.
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 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.
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.

# CA2100: Review SQL queries for security vulnerabilities

2/8/2019 • 4 minutes to read • [Edit Online](#)

TypeName	ReviewSqlQueriesForSecurityVulnerabilities
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.

## Rule description

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. 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 Framework 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](#)

Notice that this rule is violated when the `ToString` method of a type is used explicitly or implicitly to construct the query string. The following is an example.

```
int x = 10;
string query = "SELECT TOP " + x.ToString() + " FROM Table";
```

The rule is violated because a malicious user can override the `ToString()` method.

The rule also is violated when `ToString` is used 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.

## 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>

```



## See also

[Security Overview](#)

# CA2102: Catch non-CLSCompliant exceptions in general handlers

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	CatchNonClsCompliantExceptionsInGeneralHandlers
Check Id	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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewImperativeSecurity
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.

## 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 very 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 runtime 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotDeclareReadOnlyMutableReferenceTypes
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.

## Cause

An externally visible type contains an externally visible read-only field that is a mutable reference type.

## 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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	ArrayFieldsShouldNotBeReadOnly
CheckId	CA2105
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A public or protected field that holds an array is declared read-only.

## 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

```

## See also

- [System.Array](#)
- [System.Collections.ReadOnlyCollectionBase](#)

# CA2106: Secure asserts

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	SecureAsserts
CheckId	CA2106
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A method asserts a permission and performs no security checks on the caller.

## 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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	ReviewDenyAndPermitOnlyUsage
CheckId	CA2107
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A method contains a security check that specifies the PermitOnly or Deny security action.

## Rule description

The [System.Security.CodeAccessPermission.Deny](#) security action should be used only by those who have an advanced knowledge of .NET Framework 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 following 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.WriteLine("LinkDemand: ");
        }

        [EnvironmentPermissionAttribute(SecurityAction.Demand,
            Read="COMPUTERNAME;USERNAME;USERDOMAIN")]

        public static void MethodProtectedByDemand()
        {
            Console.WriteLine("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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewDeclarativeSecurityOnValueTypes
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](#).

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewVisibleEventHandlers
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	PointersShouldNotBeVisible
CheckId	CA2111
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A public or protected [System.IntPtr](#) or [System.UIntPtr](#) field is not read-only.

## 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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	SecuredTypesShouldNotExposeFields
CheckId	CA2112
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A public or protected type contains public fields and is secured by a [Link Demands](#).

## 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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	MethodSecurityShouldBeASupersetOfType
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	CallGCKeepAliveWhenUsingNativeResources
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`.

## 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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	AptcaMethodsShouldOnlyCallAptcaMethods
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.

## 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  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

2/8/2019 • 4 minutes to read • [Edit Online](#)

TypeName	AptcaTypesShouldOnlyExtendAptcaBaseTypes
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.

## 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  $T_1$  and  $T_2$  meet the following conditions, malicious callers can use the type  $T_1$  to bypass the implicit full trust inheritance demand that protects  $T_2$ :

- $T_1$  is a public type declared in a fully trusted assembly that has the APTCA attribute.
- $T_1$  inherits from a type  $T_2$  outside its assembly.
- $T_2$ '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  $T_1$ , which gives it access to inherited members declared in  $T_2$ . Because  $T_2$  does not have the APTCA attribute, its immediate derived type ( $T_1$ ) must satisfy an inheritance demand for full trust;  $T_1$  has full trust and therefore satisfies this check. The security risk is because  $x$  does not participate in satisfying the inheritance demand that protects  $T_2$  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 ( $T_1$ ) can, through programmer error, expose protected members from the type that requires full trust ( $T_2$ ). 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  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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	ReviewSuppressUnmanagedCodeSecurityUsage
CheckId	CA2118
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A public or protected type or member has the [System.Security.SuppressUnmanagedCodeSecurityAttribute](#) attribute.

## 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.

The .NET Framework 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	SealMethodsThatSatisfyPrivateInterfaces
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	SecureSerializationConstructors
Check Id	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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	StaticConstructorsShouldBePrivate
Check Id	CA2121
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A type has a static constructor that is not private.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	DoNotIndirectlyExposeMethodsWithLinkDemands
Check Id	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.

## Rule description

A link demand checks the permissions of the immediate caller only. If a member  makes no security demands of its callers, and calls code protected by a link demand, a caller without the necessary permission can use  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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	OverrideLinkDemandsShouldBeIdenticalToBase
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	WrapVulnerableFinallyClausesInOuterTry
Check Id	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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TypeLinkDemandsRequireInheritanceDemands
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	ConstantsShouldBeTransparent
Check Id	CA2130
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A constant field or an enumeration member is marked with the [SecurityCriticalAttribute](#).

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	CriticalTypesMustNotParticipateInTypeEquivalence
CheckId	CA2131
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

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.

## 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 equivalence
    [SecurityCritical]
    [TypeIdentifier("3a5b6203-2bf1-4f83-b5b4-1bcd334ad3ea", "ICriticalEquivalentInterface")]
    public interface ICriticalEquivalentInterface
    {
        void Method1();
    }

    [TypeIdentifier("3a5b6203-2bf1-4f83-b5b4-1bcd334ad3ea", "ITransparentEquivalentInterface")]
    public interface ITransparentEquivalentInterface
    {
        // CA2131 error - critical method in a type participating in equivalence
        [SecurityCritical]
        void CriticalMethod();
    }

    [SecurityCritical]
    [TypeIdentifier("3a5b6203-2bf1-4f83-b5b4-1bcd334ad3ea", "ICriticalEquivalentInterface")]
    public struct EquivalentStruct
    {
        // 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DefaultConstructorsMustHaveConsistentTransparency
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DelegatesMustBindWithConsistentTransparency
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MethodsMustOverrideWithConsistentTransparency
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.

## 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 runtime, 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 will result in a runtime [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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	SecurityRuleSetLevel2MethodsShouldNotBeProtectedWithLink Demands
Check Id	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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TransparencyAnnotationsShouldNotConflict
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.

## 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 precedence 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	TransparentMethodsMustBeVerifiable
Check Id	CA2137
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A method contains unverifiable code or returns a type by reference.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	TransparentMethodsMustNotCallSuppressUnmanagedCodeSecurityMethods
Check Id	CA2138
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A security transparent method calls a method that is marked with the [SuppressUnmanagedCodeSecurityAttribute](#) attribute.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	TransparentMethodsMustNotHandleProcessCorruptingExceptions
Check Id	CA2139
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A transparent method is marked with the [HandleProcessCorruptedStateExceptionsAttribute](#) attribute.

## 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 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. 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	TransparentMethodsMustNotReferenceCriticalCode
Check Id	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

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	TransparentMethodsMustNotSatisfyLinkDemands
Check Id	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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TransparentMethodsShouldNotBeProtectedWithLinkDemands
CheckId	CA2142
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A transparent method requires a [SecurityAction](#) or other security demand.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	TransparentMethodsShouldNotDemand
Check Id	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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	TransparentMethodsShouldNotLoadAssembliesFromByteArrays
Check Id	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](#)

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TransparentMethodsShouldNotUseSuppressUnmanagedCodeSecurity
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TypesMustBeAtLeastAsCriticalAsBaseTypes
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	SecurityTransparentCodeShouldNotAssert
CheckId	CA2147
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

Code that is marked as [SecurityTransparentAttribute](#) is not granted sufficient permissions to assert.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TransparentMethodsMustNotCallNativeCode
CheckId	CA2149
Category	Microsoft.Security
Breaking Change	Breaking

## Cause

A method calls a native function through a method stub such as P/Invoke.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	PInvokesShouldNotBeSafeCriticalFxCopRule
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.

## 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 `SecuritySafeCritical` 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 `SecuritySafeCritical` 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

2/20/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AvoidHandlingCorruptedStateExceptions
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 runtime 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

4/9/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotUseInsecureDeserializerBinaryFormatter
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 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.

## 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.JsonDataContractJsonSerializer](#)
  - [System.Web.Script.Serialization.JavaScriptSerializer](#) - Never use [System.Web.Script.Serialization.SimpleTypeResolver](#). If you must use a type resolver, you must restrict serialized types to an expected list.
  - [System.Xml.Serialization.XmlSerializer](#)
  - NewtonSoft Json.NET - Use `TypeNameHandling.None`. If you must use another value for `TypeNameHandling`, then you must restrict serialized types to an expected list.
  - Protocol Buffers
- Make the serialized data tamper proof. After serialization, cryptographically sign the serialized data. Before deserializing, validate the cryptographic signature. You must protect the cryptographic key from being disclosed, and should design for key rotations.
- Restrict serialized types. Implement a custom [System.Runtime.Serialization.SerializationBinder](#). Before deserializing with [BinaryFormatter](#), set the [Binder](#) property to an instance of your custom [SerializationBinder](#). In the overridden [BindToType](#) method, if the type is unexpected then throw an exception.

- If you restrict deserialized types, you may want to disable this rule and enable rules [CA2301](#) and [CA2302](#). Enabling rules [CA2301](#) and [CA2302](#) will help ensure that the [Binder](#) property is always set before deserializing.

## 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.
- It's safe to suppress this warning if you've taken one of the precautions above.

## 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

4/9/2019 • 3 minutes to read • [Edit Online](#)

TypeName	DoNotCallBinaryFormatterDeserializeWithoutFirstSettingBinaryFormatterBinder
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.

## Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types 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

- 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.JsonDataContractJsonSerializer](#)
  - [System.Web.Script.Serialization.JavaScriptSerializer](#) - Never use [System.Web.Script.Serialization.SimpleTypeResolver](#). If you must use a type resolver, you must restrict serialized types to an expected list.
  - [System.Xml.Serialization.XmlSerializer](#)
  - [Newtonsoft.Json.NET](#) - Use [TypeNameHandling.None](#). If you must use another value for [TypeNameHandling](#), then you must restrict serialized types to an expected list.
  - Protocol Buffers
- Make the serialized data tamper proof. After serialization, cryptographically sign the serialized data. Before deserializing, validate the cryptographic signature. You must protect the cryptographic key from being disclosed, and should design for key rotations.
- Restrict serialized types. Implement a custom [System.Runtime.Serialization.SerializationBinder](#). Before deserializing with [BinaryFormatter](#), set the [Binder](#) property to an instance of your custom [SerializationBinder](#).

In the overridden [BindToType](#) method, if the type is unexpected then throw an exception.

## 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.
- It's safe to suppress this warning if you've taken one of the precautions above.

## 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 string Author { get; set; }
    public int PageCount { 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 Author 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", "typeName");
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { 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", "typeName")
        End If
    End Function
End Class

<Serializable()
Public Class BookRecord
    Public Property Title As String
    Public Property Author 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

4/9/2019 • 3 minutes to read • [Edit Online](#)

TypeName	EnsureBinaryFormatterBinderIsSetBeforeCallingBinaryFormatterDeserialize
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.

## Rule description

Insecure deserializers are vulnerable when deserializing untrusted data. An attacker could modify the serialized data to include unexpected types 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](#) when its [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

- 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.JsonDataContractJsonSerializer](#)
  - [System.Web.Script.Serialization.JavaScriptSerializer](#) - Never use [System.Web.Script.Serialization.SimpleTypeResolver](#). If you must use a type resolver, you must restrict serialized types to an expected list.
  - [System.Xml.Serialization.XmlSerializer](#)
  - NewtonSoft Json.NET - Use [TypeNameHandling.None](#). If you must use another value for [TypeNameHandling](#), then you must restrict serialized types to an expected list.
  - Protocol Buffers
- Make the serialized data tamper proof. After serialization, cryptographically sign the serialized data. Before deserializing, validate the cryptographic signature. You must protect the cryptographic key from being disclosed, and should design for key rotations.
- Restrict serialized types. Implement a custom [System.Runtime.Serialization.SerializationBinder](#). Before deserializing with [BinaryFormatter](#), set the [Binder](#) property to an instance of your custom [SerializationBinder](#).

In the overridden [BindToType](#) method, if the type is unexpected then throw an exception.

- Ensure that all code paths have the [Binder](#) property set.

## 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.
- It's safe to suppress this warning if you've taken one of the precautions above.

## 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 string Author { get; set; }
    public int PageCount { 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);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

<Serializable()
Public Class BookRecord
    Public Property Title As String
    Public Property Author 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)
        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", "typeName");
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { 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", "typeName")
        End If
    End Function
End Class

<Serializable()
Public Class BookRecord
    Public Property Title As String
    Public Property Author 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](#)

# CA3001: Review code for SQL injection vulnerabilities

4/4/2019 • 3 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForSqlInjectionsVulnerabilities
CheckId	CA3001
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches an SQL command's text.

## 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 `.editorconfig` files.

## 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.

## 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForXssVulnerabilities
CheckId	CA3002
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches raw HTML output.

## 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 `.editorconfig` files.

## 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.

## 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForFilePathInjectionVulnerabilities
CheckId	CA3003
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches the path of a file operation.

## 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 `.editorconfig` files.

## 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.

## 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForInformationDisclosureVulnerabilities
CheckId	CA3004
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

An exception's message, stack trace, or string representation reaches web output.

## 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 `.editorconfig` files.

## 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.

# 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForLdapInjectionVulnerabilities
CheckId	CA3005
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches an LDAP statement.

## 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 `.editorconfig` files.

## 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.

## 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForProcessCommandInjectionVulnerabilities
CheckId	CA3006
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches a process command.

## 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 `.editorconfig` files.

## 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.

## 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForOpenRedirectVulnerabilities
CheckId	CA3007
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches an HTTP response redirect.

## 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 `.editorconfig` files.

## 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.

# 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForXPathInjectionVulnerabilities
CheckId	CA3008
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches an XPath query.

## 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 `.editorconfig` files.

## 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.

# 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

Type Name	ReviewCodeForXmlInjectionVulnerabilities
Check Id	CA3009
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches raw XML output.

## 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 `.editorconfig` files.

## 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.

## When to suppress warnings

Don't suppress warnings from this rule.

## 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>oscar&lt;/allowedUser>some text<allowedUser>alice</allowedUser>
    }
}

```

```

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>oscar&lt;/allowedUser>some text<allowedUser>alice</allowedUser>
    End Sub
End Class

```

# CA3010: Review code for XAML injection vulnerabilities

4/9/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForXamlInjectionVulnerabilities
CheckId	CA3010
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches a [System.Windows.Markup.XamlReader](#) Load method.

## 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 `.editorconfig` files.

## How to fix violations

Don't load untrusted XAML.

## When to suppress warnings

Don't suppress warnings from this rule.

## 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForDllInjectionVulnerabilities
CheckId	CA3011
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches a method that loads an assembly.

## 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 `.editorconfig` files.

## How to fix violations

Don't load untrusted DLLs from user input.

## When to suppress warnings

Don't suppress warnings from this rule.

## 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

4/4/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewCodeForRegexInjectionVulnerabilities
CheckId	CA3012
Category	Microsoft.Security
Breaking Change	Non Breaking

## Cause

Potentially untrusted HTTP request input reaches a regular expression.

## 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 `.editorconfig` files.

## 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.

## 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
```

# CA3075: Insecure DTD Processing

3/19/2019 • 4 minutes to read • [Edit Online](#)

TypeName	InsecureDTDProcessing
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);
            XmlTextReader reader = new XmlTextReader(tr) { DtdProcessing = DtdProcessing.Prohibit };
            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);
        }
    }
}
```

# CA3076: Insecure XSLT Script Execution

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	InsecureXSLTScriptExecution
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	InsecureDTDProcessingInAPIDesign
Check Id	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  `XmlSecureResolver` instead 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 ValidateAntiForgeryTokenToken

3/19/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkVerbHandlersWithValidateAntiForgeryToken
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;
        }
    }
}
```

# Usage Warnings

2/8/2019 • 7 minutes to read • [Edit Online](#)

Usage warnings support proper usage of the .NET Framework.

## In This Section

RULE	DESCRIPTION
<a href="#">CA1801: Review unused parameters</a>	A method signature includes a parameter that is not used in the method body.
<a href="#">CA1806: Do not ignore method results</a>	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.
<a href="#">CA1816: Call GC.SuppressFinalize correctly</a>	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).
<a href="#">CA2200: Rethrow to preserve stack details</a>	An exception is re-thrown and the exception is explicitly specified in the throw statement. If an exception is re-thrown 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.
<a href="#">CA2201: Do not raise reserved exception types</a>	This makes the original error hard to detect and debug.
<a href="#">CA2202: Do not dispose objects multiple times</a>	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.
<a href="#">CA2204: Literals should be spelled correctly</a>	A literal string in a method body contains one or more words that are not recognized by the Microsoft spelling checker library.
<a href="#">CA2205: Use managed equivalents of Win32 API</a>	A platform invoke method is defined and a method with the equivalent functionality exists in the .NET Framework class library.
<a href="#">CA2207: Initialize value type static fields inline</a>	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.
<a href="#">CA2208: Instantiate argument exceptions correctly</a>	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 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.
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	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.

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 Framework might not be usable by supported versions of the .NET Framework.
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 <code>ISerializable</code> types	To fix a violation of this rule, call the base type <code>GetObjectData</code> method or serialization constructor from the corresponding derived type method or constructor.
CA2237: Mark <code>ISerializable</code> types with <code>SerializableAttribute</code>	To be recognized by the common language runtime as serializable, types must be marked with the <code>SerializableAttribute</code> attribute even if the type uses a custom serialization routine through implementation of the <code>ISerializable</code> 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 <code>System.Runtime.Serialization.OptionalFieldAttribute</code> attribute, and the type does not provide de-serialization event handling methods.
CA2240: Implement <code>ISerializable</code> correctly	To fix a violation of this rule, make the <code>GetObjectData</code> method visible and overridable, and make sure that all instance fields are included in the serialization process or explicitly marked with the <code>NonSerializedAttribute</code> attribute.
CA2241: Provide correct arguments to formatting methods	The format argument passed to <code>System.String.Format</code> does not contain a format item that corresponds to each object argument, or vice versa.
CA2242: Test for <code>NaN</code> correctly	This expression tests a value against <code>Single.NaN</code> or <code>Double.NaN</code> . Use <code>Single.IsNaN(Single)</code> or <code>Double.IsNaN(Double)</code> 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.

# CA1801: Review unused parameters

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ReviewUnusedParameters
CheckId	CA1801
Category	Microsoft.Usage
Breaking Change	<p>Non Breaking - If the member is not visible outside the assembly, regardless of the change you make.</p> <p>Non Breaking - If you change the member to use the parameter within its body.</p> <p>Breaking - If you remove the parameter and it is visible outside the assembly.</p>

## Cause

A method signature includes a parameter that is not used in the method body. This rule does not examine the following methods:

- Methods referenced by a delegate.
- Methods used as event handlers.
- 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.

## Rule description

Review parameters in non-virtual methods that are not used in the method body to make sure no correctness 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 has to 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 for previously shipped code for which the fix would be a breaking

change.

## 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](#)

# CA1806: Do not ignore method results

2/8/2019 • 3 minutes to read • [Edit Online](#)

Type Name	DoNotIgnoreMethodResults
Check Id	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. 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.

## 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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	CallGCSuppressFinalizeCorrectly
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	RethrowToPreserveStackDetails
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotRaiseReservedExceptionTypes
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.ExecutionEngineException](#)
- [System.IndexOutOfRangeException](#)
- [System.NullReferenceException](#)
- [System.OutOfMemoryException](#)

### 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
<code>null</code> reference	<a href="#">System.ArgumentNullException</a>

PARAMETER DESCRIPTION	EXCEPTION
Outside the allowed range of values (such as an index for a collection or list)	<a href="#">System.ArgumentOutOfRangeException</a>
Invalid <code>enum</code> value	<a href="#">System.ComponentModel.InvalidEnumArgumentException</a>
Contains a format that does not meet the parameter specifications of a method (such as the format string for <code>ToString(String)</code> )	<a href="#">System.FormatException</a>
Otherwise invalid	<a href="#">System.ArgumentException</a>

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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	DoNotDisposeObjectsMultipleTimes
Check Id	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.

## 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
{
    if(stream != null)
        stream.Dispose();
}
```

## See also

- [System.IDisposable](#)
- [Dispose Pattern](#)

# CA2204: Literals should be spelled correctly

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	LiteralsShouldBeSpelledCorrectly
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	UseManagedEquivalentsOfWin32Api
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 the .NET Framework class library.

## 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 runtime 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 Function

    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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	InitializeValueTypeStaticFieldsInline
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	InstantiateArgumentExceptionsCorrectly
CheckId	CA2208
Category	Microsoft.Usage
Breaking Change	Non Breaking

## Cause

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](#).
- An incorrect string argument is passed to a parameterized constructor of an 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 `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.

## When to suppress warnings

It is safe to suppress a warning from this rule only if a parameterized constructor is called with the correct string arguments.

### Example 1

The following example shows a constructor that incorrectly instantiates an instance of the ArgumentNullException type.

```
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;
                }
            }
    };
}
```

```

using System;

namespace Samples1
{
    public class Book
    {
        private readonly string _Title;

        public Book(string title)
        {
            // Violates this rule (constructor arguments are switched)
            if (title == null)
                throw new ArgumentNullException("title cannot be a null reference (Nothing in Visual Basic)",
"title");

            _Title = title;
        }

        public string Title
        {
            get { return _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

```

## Example 2

The following example fixes the above 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;
            }
        }
    };
}
```

```
namespace Samples2
{
    public class Book
    {
        private readonly string _Title;

        public Book(string title)
        {
            if (title == null)
                throw new ArgumentNullException("title", "title cannot be a null reference (Nothing in Visual
Basic)");

            _Title = title;      }

        public string Title
        {
            get { return _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
```

# CA2211: Non-constant fields should not be visible

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	NonConstantFieldsShouldNotBeVisible
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotMarkServicedComponentsWithWebMethod
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DisposableFieldsShouldBeDisposed
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 `FT.Dispose`).

### NOTE

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.

## 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 is safe to suppress a warning from this rule if you're not responsible for releasing the resource held by the field, or if the call to [Dispose](#) occurs at a deeper calling level than the rule checks.

## 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);
        }
    }
}
```

## See also

- [System.IDisposable](#)
- [Dispose Pattern](#)

# CA2214: Do not call overridable methods in constructors

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotCallOverridableMethodsInConstructors
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 is possible that the constructor for the instance that invokes the method has not executed.

## 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, and does so 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	DisposeMethodsShouldCallBaseClassDispose
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DisposableTypesShouldDeclareFinalizer
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotMarkEnumsWithFlags
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 through static code 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

2/8/2019 • 3 minutes to read • [Edit Online](#)

Type Name	OverrideGetHashCodeOnOverridingEquals
Check Id	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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	DoNotRaiseExceptionsInExceptionClauses
Check Id	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 thrown 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	FinalizersShouldCallBaseClassFinalizer
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	FinalizersShouldBeProtected
CheckId	CA2221
Category	Microsoft.Usage
Breaking Change	Non Breaking

## Cause

A public type implements a finalizer that does not specify family (protected) access.

## Rule description

Finalizers must use the family access modifier. This rule is enforced by the C#, Visual Basic, and Visual 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotDecreaseInheritedMemberVisibility
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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MembersShouldDifferByMoreThanReturnType
CheckId	CA2223
Category	Microsoft.Usage
Breaking Change	Breaking

## Cause

Two public or protected members have signatures that are identical except for return type.

## 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_0001:  stloc.0
            IL_0002:  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

2/8/2019 • 6 minutes to read • [Edit Online](#)

Type Name	OverrideEqualsOnOverloadingOperatorEquals
Check Id	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

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	OperatorOverloadsHaveNamedAlternates
Check Id	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, and is provided for developers who program in languages that do not support overloaded operators.

This rule examines the operators listed in the following table.

C#	VISUAL BASIC	C++	ALTERNATE NAME
+ (binary)	+	+ (binary)	Add
+=	+=	+=	Add
&	And	&	BitwiseAnd
&=	And=	&=	BitwiseAnd
	Or		BitwiseOr
=	Or=	=	BitwiseOr
--	N/A	--	Decrement
/	/	/	Divide
/=	/=	/=	Divide
==	=	==	Equals

C#	VISUAL BASIC	C++	ALTERNATE NAME
^	Xor	^	Xor
^=	Xor=	^=	Xor
>	>	>	Compare
>=	> =	> =	Compare
++	N/A	++	Increment
<>	!=	Equals	
<<	<<	<<	LeftShift
<<=	<<=	<<=	LeftShift
<	<	<	Compare
<=	< =	< =	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
+ (unary)	N/A	+	Plus

C#	VISUAL BASIC	C++	ALTERNATE NAME
false	IsFalse	False	IsTrue (Property)

N/A == Cannot be overloaded in the selected language.

The rule also checks implicit and explicit cast operators in a type (`SomeType`) by checking for methods named `ToSomeType` and `FromSomeType`.

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 are implementing a shared library. Applications can ignore a warning from this rule.

## Configurability

If you're running this rule from [FxCop analyzers](#) (and not through static code 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	OperatorsShouldHaveSymmetricalOverloads
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 through static code 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

2/8/2019 • 3 minutes to read • [Edit Online](#)

TypeName	CollectionPropertiesShouldBeReadOnly
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'), 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	DoNotShipUnreleasedResourceFormats
CheckId	CA2228
Category	Microsoft.Usage
Breaking Change	Non Breaking

## Cause

A resource file was built using a version of the .NET Framework that is not currently supported.

## Rule description

Resource files that were built by using pre-release versions of the .NET Framework might not be usable by supported versions of the .NET Framework.

## How to fix violations

To fix a violation of this rule, build the resource using a supported version of the .NET Framework.

## When to suppress warnings

Do not suppress a warning from this rule.

# CA2229: Implement serialization constructors

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ImplementSerializationConstructors
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 [System.Runtime.Serialization.SerializationInfo](#) object and a [System.Runtime.Serialization.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 re-create objects that have been serialized using the [System.Runtime.Serialization.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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	UseParamsForVariableArguments
Check Id	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.

## 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 Visual 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	OverloadOperatorEqualsOnOverridingValueTypeEquals
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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkWindowsFormsEntryPointsWithStaThread
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

3/12/2019 • 2 minutes to read • [Edit Online](#)

TypeName	OperationsShouldNotOverflow
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.

## 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

3/12/2019 • 2 minutes to read • [Edit Online](#)

Type Name	PassSystemUriObjectsInsteadOfStrings
Check Id	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 through static code 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkAllNonSerializableFields
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 is not serializable.

An exception to this is when the type uses custom serialization via the [System.Runtime.Serialization.ISerializable](#) interface. Types implementing this interface provide their own serialization logic, and so CA2235 will not fire for non-serializable instance fields of such types.

## 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 a type that violates the rule and a type 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

```

## Related rules

[CA2236: Call base class methods on ISerializable types](#)

[CA2240: Implement ISerializable correctly](#)

[CA2229: Implement serialization constructors](#)

[CA2238: Implement serialization methods correctly](#)

CA2237: Mark ISerializable types with SerializableAttribute

CA2239: Provide deserialization methods for optional fields

CA2120: Secure serialization constructors

# CA2236: Call base class methods on ISerializable types

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	CallBaseClassMethodsOnISerializableTypes
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	MarkISerializableTypesWithSerializable
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ImplementSerializationMethodsCorrectly
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](#)



# CA2239: Provide deserialization methods for optional fields

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	ProvideDeserializationMethodsForOptionalFields
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

2/8/2019 • 5 minutes to read • [Edit Online](#)

Type Name	ImplementISerializableCorrectly
Check Id	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
    End Class

```

```

        <SecurityPermissionAttribute(SecurityAction.LinkDemand,
Flags:=SecurityPermissionFlag.SerializationFormatter)> _
Protected Overrides 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

Type Name	ProvideCorrectArgumentsToFormattingMethods
Check Id	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.

## 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 runtime.

## 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.

## 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	TestForNaNCorrectly
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

2/8/2019 • 2 minutes to read • [Edit Online](#)

TypeName	AttributeStringLiteralsShouldParseCorrectly
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](#)

# Code Analysis Policy Errors

2/8/2019 • 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](#)

# C++ Core Guidelines Checker Reference

4/16/2019 • 7 minutes to read • [Edit Online](#)

This section lists C++ Core Guidelines Checker warnings. For information about Code Analysis, see [/analyze \(Code Analysis\)](#) and [Quick Start: Code Analysis for C/C++](#).

## NOTE

Some warnings belong to more than one group, and not all warnings have a complete reference topic.

## OWNER\_POINTER Group

[C26402 DONT\\_HEAP\\_ALLOCATE\\_MOVABLE\\_RESULT](#) Return a scoped object instead of a heap-allocated if it has a move constructor. See [C++ Core Guidelines R.3](#).

[C26403 RESET\\_OR\\_DELETE\\_OWNER](#) Reset or explicitly delete an owner<T> pointer '%variable%'. See [C++ Core Guidelines R.3](#).

[C26404 DONT\\_DELETE\\_INVALID](#) Do not delete an owner<T> that may be in invalid state. See [C++ Core Guidelines R.3](#).

[C26405 DONT\\_ASSIGN\\_TO\\_VALID](#) Do not assign to an owner<T> that may be in valid state. See [C++ Core Guidelines R.3](#).

[C26406 DONT\\_ASSIGN\\_RAW\\_TO\\_OWNER](#) Do not assign a raw pointer to an owner<T>. See [C++ Core Guidelines R.3](#).

[C26407 DONT\\_HEAP\\_ALLOCATE\\_UNNECESSARILY](#) Prefer scoped objects, don't heap-allocate unnecessarily. See [C++ Core Guidelines R.5](#).

[C26429 USE\\_NOTNULL](#) Symbol '%symbol%' is never tested for nullness, it can be marked as `not_null`. See [C++ Core Guidelines F.23](#).

[C26430 TEST\\_ON\\_ALL\\_PATHS](#) Symbol '%symbol%' is not tested for nullness on all paths. See [C++ Core Guidelines F.23](#).

[C26431 DONT\\_TEST\\_NOTNULL](#) The type of expression '%expr%' is already `gsl::not_null`. Do not test it for nullness. See [C++ Core Guidelines F.23](#).

## RAW\_POINTER Group

[C26400 NO\\_RAW\\_POINTER\\_ASSIGNMENT](#) Do not assign the result of an allocation or a function call with an owner<T> return value to a raw pointer; use owner<T> instead. See [C++ Core Guidelines I.11](#).

[C26401 DONT\\_DELETE\\_NON\\_OWNER](#) Do not delete a raw pointer that is not an owner<T>. See [C++ Core Guidelines I.11](#).

[C26402 DONT\\_HEAP\\_ALLOCATE\\_MOVABLE\\_RESULT](#) Return a scoped object instead of a heap-allocated if it has a move constructor. See [C++ Core Guidelines R.3](#).

[C26408 NO\\_MALLOC\\_FREE](#) Avoid `malloc()` and `free()`, prefer the nothrow version of `new` with `delete`. See [C++ Core Guidelines R.10](#).

[C26409 NO\\_NEW\\_DELETE](#) Avoid calling `new` and `delete` explicitly, use `std::make_unique<T>` instead. See [C++ Core Guidelines R.10](#).

## Core Guidelines R.11.

C26429 USE\_NONNULL Symbol '%symbol%' is never tested for nullness, it can be marked as `not_null`. See C++ Core Guidelines F.23.

C26430 TEST\_ON\_ALL\_PATHS Symbol '%symbol%' is not tested for nullness on all paths. See C++ Core Guidelines F.23.

C26431 DONT\_TEST\_NONNULL The type of expression '%expr%' is already `gsl::not_null`. Do not test it for nullness. See C++ Core Guidelines F.23.

C26481 NO\_POINTER\_ARITHMETIC Don't use pointer arithmetic. Use `span` instead. See C++ Core Guidelines Bounds.1.

C26485 NO\_ARRAY\_TO\_POINTER\_DECAY Expression '%expr%': No array to pointer decay. See C++ Core Guidelines Bounds.3.

## UNIQUE\_POINTER Group

C26410 NO\_REF\_TO\_CONST\_UNIQUE\_PTR The parameter '%parameter%' is a reference to `const` unique pointer, use `const T*` or `const T&` instead. See C++ Core Guidelines R.32.

C26411 NO\_REF\_TO\_UNIQUE\_PTR The parameter '%parameter%' is a reference to unique pointer and it is never reassigned or reset, use `T*` or `T&` instead. See C++ Core Guidelines R.33.

C26414 RESET\_LOCAL\_SMART\_PTR Move, copy, reassign, or reset a local smart pointer '%symbol%'. See C++ Core Guidelines R.5.

C26415 SMART\_PTR\_NOT\_NEEDED Smart pointer parameter '%symbol%' is used only to access contained pointer. Use `T*` or `T&` instead. See C++ Core Guidelines R.30.

## SHARED\_POINTER Group

C26414 RESET\_LOCAL\_SMART\_PTR Move, copy, reassign, or reset a local smart pointer '%symbol%'. See C++ Core Guidelines R.5.

C26415 SMART\_PTR\_NOT\_NEEDED Smart pointer parameter '%symbol%' is used only to access contained pointer. Use `T*` or `T&` instead. See C++ Core Guidelines R.30.

C26416 NO\_RVALUE\_REF\_SHARED\_PTR Shared pointer parameter '%symbol%' is passed by rvalue reference. Pass by value instead. See C++ Core Guidelines R.34.

C26417 NO\_LVALUE\_REF\_SHARED\_PTR Shared pointer parameter '%symbol%' is passed by reference and not reset or reassigned. Use `T*` or `T&` instead. See C++ Core Guidelines R.35.

C26418 NO\_VALUE\_OR\_CONST\_REF\_SHARED\_PTR Shared pointer parameter '%symbol%' is not copied or moved. Use `T*` or `T&` instead. See C++ Core Guidelines R.36.

## DECLARATION Group

C26426 NO\_GLOBAL\_INIT\_CALLS Global initializer calls a non-`constexpr` function '%symbol%'. See C++ Core Guidelines I.22.

C26427 NO\_GLOBAL\_INIT\_EXTERN Global initializer accesses extern object '%symbol%'. See C++ Core Guidelines I.22.

C26444 NO\_UNNAMED\_RAIIL\_OBJECTS Avoid unnamed objects with custom construction and destruction. See ES.84: Don't (try to) declare a local variable with no name.

## CLASS Group

C26432 **DEFINE\_OR\_DELETE\_SPECIAL\_OPS** If you define or delete any default operation in the type '%symbol%', define or delete them all. See [C++ Core Guidelines C.21](#).

C26433 **OVERRIDE\_EXPLICITLY** Function '%symbol%' should be marked with 'override'. See [C.128: Virtual functions should specify exactly one of virtual, override, or final](#).

C26434 **DONT\_HIDE\_METHODS** Function '%symbol\_1%' hides a non-virtual function '%symbol\_2%'. See [C++ Core Guidelines C.128](#).

C26435 **SINGLE\_VIRTUAL\_SPECIFICATION** Function '%symbol%' should specify exactly one of 'virtual', 'override', or 'final'. See [C.128: Virtual functions should specify exactly one of virtual, override, or final](#).

C26436 **NEED\_VIRTUAL\_DTOR** The type '%symbol%' with a virtual function needs either public virtual or protected nonvirtual destructor. See [C++ Core Guidelines C.35](#).

C26443 **NO\_EXPLICIT\_DTOR\_OVERRIDE** Overriding destructor should not use explicit 'override' or 'virtual' specifiers. See [C.128: Virtual functions should specify exactly one of virtual, override, or final](#).

## TYPE Group

C26437 **DONT\_SLICE** Do not slice. See [C++ Core Guidelines ES.63](#).

## STYLE Group

C26438 **NO\_GOTO** Avoid `goto`. See [C++ Core Guidelines ES.76](#).

## FUNCTION Group

C26439 **SPECIAL\_NOEXCEPT** This kind of function may not throw. Declare it `noexcept`. See [C++ Core Guidelines F.6](#).

C26440 **DECLARE\_NOEXCEPT** Function '%symbol%' can be declared `noexcept`. See [C++ Core Guidelines F.6](#).

C26447 **DONT\_THROW\_IN\_NOEXCEPT** The function is declared **noexcept** but calls a function which may throw exceptions. See [C++ Core Guidelines: F.6: If your function may not throw, declare it noexcept](#).

## CONCURRENCY Group

C26441 **NO\_UNNAMED GUARDS** Guard objects must be named. See [C++ Core Guidelines cp.44](#).

## CONST Group

C26460 **USE\_CONST\_REFERENCE\_ARGUMENTS** The reference argument '%argument%' for function '%function%' can be marked as `const`. See [C++ Core Guidelines con.3](#).

C26461 **USE\_CONST\_POINTER\_ARGUMENTS**: The pointer argument '%argument%' for function '%function%' can be marked as a pointer to `const`. See [C++ Core Guidelines con.3](#).

C26462 **USE\_CONST\_POINTER\_FOR\_VARIABLE** The value pointed to by '%variable%' is assigned only once, mark it as a pointer to `const`. See [C++ Core Guidelines con.4](#).

C26463 **USE\_CONST\_FOR\_ELEMENTS** The elements of array '%array%' are assigned only once, mark elements `const`. See [C++ Core Guidelines con.4](#).

C26464 **USE\_CONST\_POINTER\_FOR\_ELEMENTS** The values pointed to by elements of array '%array%' are assigned only once, mark elements as pointer to `const`. See [C++ Core Guidelines con.4](#).

C26496 USE\_CONST\_FOR\_VARIABLE The variable '%variable%' is assigned only once, mark it as `const`. See C++ Core Guidelines con.4.

C26497 USE\_CONSTEXPR\_FOR\_FUNCTION This function %function% could be marked `constexpr` if compile-time evaluation is desired. See C++ Core Guidelines F.4.

C26498 USE\_CONSTEXPR\_FOR\_FUNCTIONCALL This function call %function% can use `constexpr` if compile-time evaluation is desired. See C++ Core Guidelines con.5.

## TYPE Group

C26465 NO\_CONST\_CAST\_UNNECESSARY Don't use `const_cast` to cast away `const`. `const_cast` is not required; constness or volatility is not being removed by this conversion. See C++ Core Guidelines Type.3.

C26466 NO\_STATIC\_DOWNCASE\_POLYMORPHIC Don't use `static_cast` downcasts. A cast from a polymorphic type should use `dynamic_cast`. See C++ Core Guidelines Type.2.

C26471 NO\_REINTERPRET\_CAST\_FROM\_VOID\_PTR Don't use `reinterpret_cast`. A cast from `void*` can use `static_cast`. See C++ Core Guidelines Type.1.

C26472 NO\_CASTS\_FOR\_ARITHMETIC\_CONVERSION Don't use a `static_cast` for arithmetic conversions. Use brace initialization, `gsl::narrow_cast`, or `gsl::narrow`. See C++ Core Guidelines Type.1.

C26473 NO\_IDENTITY\_CAST Don't cast between pointer types where the source type and the target type are the same. See C++ Core Guidelines Type.1.

C26474 NO\_IMPLICIT\_CAST Don't cast between pointer types when the conversion could be implicit. See C++ Core Guidelines Type.1.

C26475 NO\_FUNCTION\_STYLE\_CASTS Do not use function style C-casts. See C++ Core Guidelines ES.49.

C26490 NO\_REINTERPRET\_CAST Don't use `reinterpret_cast`. See C++ Core Guidelines Type.1.

C26491 NO\_STATIC\_DOWNCASE Don't use `static_cast` downcasts. See C++ Core Guidelines Type.2.

C26492 NO\_CONST\_CAST Don't use `const_cast` to cast away `const`. See C++ Core Guidelines Type.3.

C26493 NO\_CSTYLE\_CAST Don't use C-style casts. See C++ Core Guidelines Type.4.

C26494 VAR\_USE\_BEFORE\_INIT Variable '%variable%' is uninitialized. Always initialize an object. See C++ Core Guidelines Type.5.

C26495 MEMBER\_UNINIT Variable '%variable%' is uninitialized. Always initialize a member variable. See C++ Core Guidelines Type.6.

## BOUNDS Group

C26446 USE GSL\_AT Prefer to use `gsl::at()` instead of unchecked subscript operator. See C++ Core Guidelines: Bounds.4: Don't use standard-library functions and types that are not bounds-checked.

C26481 NO\_POINTER\_ARITHMETIC Don't use pointer arithmetic. Use `span` instead. See C++ Core Guidelines Bounds.1

C26482 NO\_DYNAMIC\_ARRAY\_INDEXING Only index into arrays using constant expressions. See C++ Core Guidelines Bounds.2

C26483 STATIC\_INDEX\_OUT\_OF\_RANGE Value %value% is outside the bounds (0, %bound%) of variable '%variable%'. Only index into arrays using constant expressions that are within bounds of the array. See C++ Core Guidelines Bounds.2

C26485 NO\_ARRAY\_TO\_POINTER\_DECAY Expression '%expr%': No array to pointer decay. See [C++ Core Guidelines](#) Bounds.3

## GSL Group

C26445 NO\_SPAN\_REF A reference to `gsl::span` or `std::string_view` may be an indication of a lifetime issue. See [C++ Core Guidelines](#) GSL.view: Views

C26446 USE\_GSL\_AT Prefer to use `gsl::at()` instead of unchecked subscript operator. See [C++ Core Guidelines](#): Bounds.4: Don't use standard-library functions and types that are not bounds-checked.

C26448 USE\_GSL\_FINALLY Consider using `gsl::finally` if final action is intended. See [C++ Core Guidelines](#): GSL.util: Utilities.

C26449 NO\_SPAN\_FROM\_TEMPORARY `gsl::span` or `std::string_view` created from a temporary will be invalid when the temporary is invalidated. See [C++ Core Guidelines](#): GSL.view: Views.

## Deprecated Warnings

The following warnings are present in an early experimental rule set of the core guidelines checker, but are now deprecated and can be safely ignored. The warnings are superceded by warnings from the list above.

- 26412 Deref\_Invalid\_Pointer
- 26413 Deref\_Nullptr
- 26420 Assign\_NonOwner\_To\_Explicit\_Owner
- 26421 Assign\_Valid\_Owner
- 26422 Valid\_Owner\_Leaving\_Scope
- 26423 Allocation\_Not\_Assigned\_To\_Owner
- 26424 Valid\_Allocation\_Leaving\_Scope
- 26425 Assigning\_To\_Static
- 26499 No\_Lifetime\_Tracking

## See Also

[Using the C++ Core Guidelines Checkers](#)

# C26400 NO\_RAW\_POINTER\_ASSIGNMENT

3/21/2019 • 2 minutes to read • [Edit Online](#)

This check helps to enforce the *rule I.11: Never transfer ownership by a raw pointer (`T*`)*, which is a subset of the rule *R.3: A raw pointer (a `T*`) is non-owning*. Specifically, it warns on any call to operator `new` which saves its result in a variable of raw pointer type. It also warns on calls to functions that return `gsl::owner<T>` if their results are assigned to raw pointers. The idea here is that you should clearly state ownership of memory resources. For more information, see the [C++ Core Guidelines](#).

The easiest way to fix this is to use `auto` declaration if the resource is assigned immediately at the variable declaration. If this is not possible, then we suggest that you use the type `gsl::owner<T>`. The `auto` declarations initialized with operator `new` are "owners" because we assume that the result of any allocation is implicitly an owner pointer. We transfer this assumption to the `auto` variable and treat it as `owner<T>`.

If this check flags a call to a function which returns `owner<T>`, this may be an indication of a legitimate bug in code. Basically, it points to a place where the code leaks an explicit notion of ownership (and maybe the resource itself).

## Remarks

This rule currently checks only local variables. If allocation is assigned to a formal parameter, global variable, class member, and so on, it is not flagged. Appropriate coverage of such scenarios is a part of future work.

## Example 1: Simple allocation

```
char *buffer = nullptr;
if (useCache)
    buffer = GetCache();
else
    buffer = new char[bufferSize]; // C26400
```

## Example 2: Simple allocation (fixed with `gsl::owner<T>`)

```
gsl::owner<char*> buffer = nullptr;
if (useCache)
    buffer = GetCache();
else
    buffer = new char[bufferSize]; // OK

Example 3: Simple allocation (fixed with auto)
auto buffer = useCache ? GetCache() : new char[bufferSize]; // OK
```

# C26401 DONT\_DELETE\_NON\_OWNER

4/16/2019 • 2 minutes to read • [Edit Online](#)

This check detects places where moving to `owner<T>` can be a good option for the first stage of refactoring. Like C26400 it enforces rules I.11 and R.3, but focuses on the "release" portion of the pointer lifetime. It warns on any call to operator `delete` if its target is neither an `owner<T>` nor an implicitly assumed owner. For more information, see [C26400](#) regarding the auto declarations. This does include expressions that refer to global variables, formals, and so on.

Warnings C26400 and C26401 always occur with [C26409](#), but they are more appropriate for scenarios where immediate migration to smart pointers is not feasible. In such cases the `owner<T>` concept can be adopted first and C26409 may be temporarily suppressed.

# C26401 DONT\_DELETE\_NON\_OWNER

4/16/2019 • 2 minutes to read • [Edit Online](#)

This check detects places where moving to `owner<T>` can be a good option for the first stage of refactoring. Like C26400 it enforces rules I.11 and R.3, but focuses on the "release" portion of the pointer lifetime. It warns on any call to operator `delete` if its target is neither an `owner<T>` nor an implicitly assumed owner. For more information, see [C26400](#) regarding the auto declarations. This does include expressions that refer to global variables, formals, and so on.

Warnings C26400 and C26401 always occur with [C26409](#), but they are more appropriate for scenarios where immediate migration to smart pointers is not feasible. In such cases the `owner<T>` concept can be adopted first and C26409 may be temporarily suppressed.

# C26403 RESET\_OR\_DELETE\_OWNER

3/1/2019 • 2 minutes to read • [Edit Online](#)

Owner pointers are like unique pointers: they own a resource exclusively, and manage release of the resource, as well as its transfers to other owners. This check validates that a local owner pointer properly maintains its resource through all execution paths in a function. If the resource was not transferred to another owner, or was not explicitly released, the checker warns, and points to the declaration of the pointer variable.

For more information, see the [C++ Core Guidelines](#).

## Remarks

- Currently this check doesn't give the exact path which fails to release the resource. This behavior may be improved in future releases. It may be difficult to find exact location for a fix. The better approach is to try to replace plain pointers in complex functions with unique pointers to avoid any risks.
- The check may discard an over-complicated function in order to not block code analysis. Generally, the complexity of functions should be maintained under some reasonable threshold. We may consider adding a local complexity check to the C++ Core Guidelines module if there is clear demand for it. This limitation is applicable to other rules which are sensitive to data flow.
- The warning may fire on clearly false positive cases where memory is deleted only after the nullness check of a pointer. This is the result of a current limitation of the tool's API, but it may be improved in future.

## Example 1: Missing cleanup during error handling

```
gsl::owner<int*> sequence = GetRandomSequence(); // C26403

try
{
    StartSimulation(sequence);
}
catch (const std::exception& e)
{
    if (KnownException(e))
        return; // Skipping the path which deletes the owner.

    ReportException(e);
}

delete [] sequence;
```

# C26404 DONT\_DELETE\_INVALID

2/8/2019 • 2 minutes to read • [Edit Online](#)

Once owner pointer releases or transfers its resource, it gets into an "invalid" state. Deleting such a pointer may lead to immediate memory corruption due to double delete, or to an access violation when the deleted resource is accessed from another owner pointer.

## Example 1: Deleting an owner after transferring its value

```
gsl::owner<State*> validState = nullptr;
gsl::owner<State*> state = ReadState();
validState = state;
if (!IsValid(state))
    delete state; // C26404
```

## Example 2: Deleting an uninitialized owner

```
gsl::owner<Message*> message;
if (popLast)
    message = ReleaseMessage();
delete message; // C26404
```

# C26405 DONT\_ASSIGN\_TO\_VALID

2/8/2019 • 2 minutes to read • [Edit Online](#)

If an owner pointer already points to a valid memory buffer, it must not be assigned to another value without releasing its current resource first. Such assignment may lead to a resource leak even if the resource address is copied into some raw pointer (because raw pointers shouldn't release resources).

## Example 1: Overwriting an owner in a loop

```
gsl::owner<Shape*> shape = nullptr;
while (shape = NextShape()) // C26405
    Process(shape) ? delete shape : 0;
```

# C26406 DONT\_ASSIGN\_RAW\_TO\_OWNER

2/8/2019 • 2 minutes to read • [Edit Online](#)

Owners are initialized from allocations or from other owners. Assigning a value from a raw pointer to an owner pointer is not allowed. Raw pointers don't guarantee ownership transfer; there is still may be an original owner which holds the resource and will attempt to release it. Note that assigning a value from owner to a raw pointer is fine; raw pointers are valid clients to access resources, but not to manage them.

## Example 1: Using address of object

```
gsl::owner<Socket*> socket = defaultSocket ? &defaultSocket : new Socket(); // C26406
```

# C26407 DONT\_HEAP\_ALLOCATE\_UNNECESSARILY

2/8/2019 • 2 minutes to read • [Edit Online](#)

To avoid unnecessary use of pointers we try to detect common patterns of local allocations, for example when the result of a call to operator new is stored in a local variable and later explicitly deleted. This supports the rule R.5: *Prefer scoped objects, don't heap-allocate unnecessarily*. The suggested fix is to use an RAII type instead of a raw pointer and allow it to deal with resources. If an allocation is a single object, then it may be obviously unnecessary and a local variable of the object's type would work better.

## Remarks

- To reduce the number of warnings, this pattern is detected for owner pointers only. So, it is necessary to mark owners properly first. We can easily extend this to cover raw pointers if we receive feedback from customers in support of such scenario.
- The scoped object term may be a bit misleading, but the general idea is that we suggest using either a local variable whose lifetime is automatically managed, or a smart object which efficiently manages dynamic resources. Smart objects can of course do heap allocations, but it is not explicit in the code.
- If the warning fires on array allocation (which is usually needed for dynamic buffers), the fix can be to use standard containers, or `std::unique_pointer<T[]>`.
- The pattern is detected only for local variables, so we don't warn on cases where an allocation is assigned to, say, a global variable and then deleted in the same function.

## Example 1: Unnecessary object allocation on heap

```
auto tracer = new Tracer();
ScanObjects(tracer);
delete tracer; // C26407
```

## Example 2: Unnecessary object allocation on heap (fixed with local object)

```
Tracer tracer; // OK
ScanObjects(&tracer);
```

## Example 3: Unnecessary buffer allocation on heap

```
auto value = new char[maxValueSize];
if (ReadSetting(name, value, maxValueSize))
    CheckValue(value);
delete[] value; // C26407
```

## Example 4: Unnecessary buffer allocation on the heap (fixed with container)

```
auto value = std::vector<char>(maxLengthSize); // OK
if (ReadSetting(name, value.data(), maxLengthSize))
    CheckValue(value.data());
```

# C26408 NO\_MALLOC\_FREE

2/8/2019 • 2 minutes to read • [Edit Online](#)

This warning flags places where `malloc` or `free` is invoked explicitly in accordance to R.10: Avoid `malloc` and `free`. One potential fix for such warnings would be to use `std::make_unique` to avoid explicit creation and destruction of objects. If such a fix is not acceptable, operator `new` and `delete` should be preferred. In some cases, if exceptions are not welcome, `malloc` and `free` can be replaced with the nothrow version of operators `new` and `delete`.

## Remarks

- To detect malloc() we check if a call invokes a global function with name "malloc" or "std::malloc". The function must return a pointer to `void` and accept one parameter of unsigned integral type.
- To detect free() we check global functions with names "free" or "std::free" which return no result and accept one parameter, which is a pointer to `void`.

# C26409 NO\_NEW\_DELETE

2/8/2019 • 2 minutes to read • [Edit Online](#)

Even if code is clean of calls to malloc() and free() we still suggest that you consider better options than explicit use of operators [new](#) and [delete](#). See more details in the description of the rule *R.11: Avoid calling new and delete explicitly*. The ultimate fix is to start using smart pointers with appropriate factory functions, such as [std::make\\_unique](#).

## Remarks

- The checker warns on calls to any kind of operator `new` or `delete`: scalar, vector, overloaded versions (global and class-specific), as well as on placement versions. The latter case may require some clarifications on the Core Guidelines in terms of suggested fixes and may be omitted in the future.

# C26410 NO\_REF\_TO\_CONST\_UNIQUE\_PTR

2/8/2019 • 2 minutes to read • [Edit Online](#)

Generally, references to const unique pointer are meaningless. They can safely be replaced by a raw reference or a pointer.

## Remarks

- Unique pointer checks have rather broad criteria to identify smart pointers. The rule R.31: *If you have non-std smart pointers, follow the basic pattern from std describes the unique pointer and shared pointer concepts.* The heuristic is simple, but may lead to surprises: a smart pointer type is any type which defines either operator-> or operator\*; a copy-able type (shared pointer) must have either public copy constructor or overloaded assignment operator which deals with a non-R-value reference parameter.
- Template code may produce a lot of noise. Keep in mind that templates can be instantiated with various type parameters with different levels of indirection, including references. Some warnings may not be obvious and fixes may require some rework of templates (for example, explicit removal of reference indirection). If template code is intentionally generic, the warning can be suppressed.

## Example 1: Unnecessary reference

```
std::vector<std::unique_ptr<Tree>> roots = GetRoots();
std::for_each(
    roots.begin(),
    roots.end(),
    [](const auto &root) { Rebalance(root.get()); }); // C26410
```

# C26411 NO\_REF\_TO\_UNIQUE\_PTR

3/1/2019 • 2 minutes to read • [Edit Online](#)

Passing a unique pointer by reference assumes that its resource may be released or transferred inside of a target function. If function uses its parameter only to access the resource, it is safe to pass a raw pointer or a reference.

## Remarks

- The limitations from the warning [C26410](#) are applicable here as well.
- The heuristic to detect "release" or "reset" access to the unique pointer is rather naive: we only detect calls to assignment operators and functions named "reset" (case-insensitive). Obviously, this detection doesn't cover all possible cases of smart pointer modifications (for example, std::swap, or any special non-const function in a custom smart pointer). It is expected that this warning will produce many false positives on custom types, as well as in some scenarios dealing with standard unique pointers. The heuristic will be improved as we implement more checks focused on smart pointers.
- The fact that smart pointers are often templates brings an interesting limitation related to the fact that the compiler is not required to process template code in templates if it's not used. In some minimal code bases that have limited use of smart pointer interfaces, the checker may produce unexpected results due to its inability to properly identify semantics of the template type (because some important functions may never be used). For the standard `unique_ptr`, this limitation is mitigated by recognizing the type's name. This may be extended in future to cover more well-known smart pointers.
- Lambda expressions with implicit capture-by-reference may lead to surprising warnings about references to unique pointers. Currently all captured reference parameters in lambdas are reported, regardless of whether they are reset or not. The heuristic here will be extended to correlate lambda fields with lambda parameters in a future release.

## Example: Unnecessary reference

```
void TraceValid(std::unique_ptr<Slot> &slot)    // C26411
{
    if (!IsDamaged(slot.get()))
        std::cout << *slot.get();
}

void ReleaseValid(std::unique_ptr<Slot> &slot)  // OK
{
    if (!IsDamaged(slot.get()))
        slot.reset(nullptr);
}
```

# C26414 RESET\_LOCAL\_SMART\_PTR

2/27/2019 • 2 minutes to read • [Edit Online](#)

"Move, copy, reassign or reset a local smart pointer."

**C++ Core Guidelines:** R.5: Prefer scoped objects, don't heap-allocate unnecessarily

Smart pointers are convenient for dynamic resource management, but they are not always necessary. For example, creating of a local dynamic buffer can be easily (and sometimes more efficiently) managed by standard containers. For single objects it may be unnecessary to do dynamic allocation at all (e.g. if such objects never outlive their creator function) and they can be replaced with local variables. Smart pointers become handy when scenario requires changing of ownership, i.e. reassigning of a dynamic resource multiple times or in multiple paths. This also includes cases where resources are obtained from external code and smart pointers are used to extend resource's lifetime.

## Remarks

- In addition to the standard std::unique\_pointer and std::shared\_pointer templates, this check recognizes user defined types which are likely intended to be smart pointers. Such types are expected to define the following operations:
  - overloaded dereference or member access operators, that are public and not marked as deleted;
  - public destructor which is neither deleted nor defaulted. This includes destructors which are explicitly defined empty.
  - The type Microsoft::WRL::ComPtr behaves as a shared pointer, but it is often used in quite specific scenarios which are affected by the COM lifetime management. To avoid excessive noise this type is filtered out.
  - This check looks for explicit local allocations assigned to smart pointers to identify if scoped variables could work as an alternative. In addition to direct calls to operator new, special functions like std::make\_unique and std::make\_shared are also interpreted as direct allocations.

## Example

dynamic buffer

```
void unpack_and_send(const frame &f)
{
    auto buffer = std::make_unique<char[]>(f.size()); // C26414
    f.unpack(buffer.get());
    // ...
}
```

dynamic buffer – replaced by container

```
void unpack_and_send(const frame &f)
{
    auto buffer = std::vector<char>(f.size());
    f.unpack(buffer.data());
    // ...
}
```

# C26415 SMART\_PTR\_NOT\_NEEDED

2/8/2019 • 2 minutes to read • [Edit Online](#)

"Smart pointer parameter is used only to access contained pointer. Use T\* or T& instead."

**C++ Core Guidelines:** R.30: Take smart pointers as parameters only to explicitly express lifetime semantics

Using a smart pointer type to pass data to a function indicates that the target function needs to manage the lifetime of the contained object. However, if the function only uses the smart pointer to access the contained object and never actually calls any code that may lead to its deallocation (that is, never affect its lifetime), there is usually no need to complicate the interface with smart pointers. A plain pointer or reference to the contained object is preferred.

## Remarks

This check covers a majority of scenarios that also cause C26410, C26415, C26417, and C26418. It is better to clean up SMART\_PTR\_NOT\_NEEDED first and then switch to edge cases for shared or unique pointers. For more focused cleanup, this warning can be disabled.

In addition to the standard std::unique\_pointer and std::shared\_pointer templates, this check recognizes user-defined types that are likely intended to be smart pointers. Such types are expected to define the following operations:

- Overloaded dereference or member access operators that are public and not marked as deleted.
- Public destructor that's not deleted or defaulted. This includes destructors that are explicitly defined empty.

Interpretation of the operations that can affect the lifetime of contained objects is broad and includes:

- Any function that accepts a pointer or reference parameter to a non-constant smart pointer
- Copy or move constructors or assignment operators
- Non-constant functions

## Example

Cumbersome lifetime management.

```
bool set_initial_message(
    const std::unique_ptr<message> &m) // C26415, also C26410 NO_REF_TO_CONST_UNIQUE_PTR
{
    if (!m || initial_message_)
        return false;

    initial_message_.reset(m.get());
    return true;
}

void pass_message(const message_info &info)
{
    auto m = std::make_unique<message>(info);
    const auto release = set_initial_message(m);
    // ...
    if (release)
        m.release();
}
```

## Example

Cumbersome lifetime management - reworked.

```
void set_initial_message(std::shared_ptr<message> m) noexcept
{
    if (m && !initial_message_)
        initial_message_ = std::move(m);
}

void pass_message(const message_info &info)
{
    auto m = std::make_shared<message>(info);
    set_initial_message(m);
    // ...
}
```

# C26416 NO\_RVALUE\_REF\_SHARED\_PTR

3/21/2019 • 2 minutes to read • [Edit Online](#)

"Shared pointer parameter is passed by rvalue reference. Pass by value instead."

**C++ Core Guidelines:** R.34: Take a shared\_ptr<widget> parameter to express that a function is part owner

Passing a shared pointer by rvalue reference is usually unnecessary. Unless it is an implementation of move semantics for a shared pointer type itself, shared pointer objects can be safely passed by value. Using rvalue reference may be also an indication that unique pointer is more appropriate since it clearly transfers unique ownership from caller to callee.

## Remarks

- This check recognizes std::shared\_pointer and user-defined types which are likely to behave like shared pointers.  
The following traits are expected for user-defined shared pointers:
  - overloaded dereference or member access operators (public and non-deleted);
  - copy constructor or copy assignment operator (public and non-deleted);
  - public destructor which is neither deleted nor defaulted. Empty destructors are still counted as user-defined.

## Example

questionable constructor optimization

```
action::action(std::shared_ptr<transaction> &&t) noexcept // C26416
    : transaction_(std::move(t))
{}

action::action(std::shared_ptr<transaction> &t) noexcept // also C26417 LVALUE_REF_SHARED_PTR
    : transaction_(t)
{}
```

## Example

questionable constructor optimization - simplified

```
action::action(std::shared_ptr<transaction> t) noexcept
    : transaction_(std::move(t))
{}
```

# C26417 NO\_LVALUE\_REF\_SHARED\_PTR

3/21/2019 • 2 minutes to read • [Edit Online](#)

"Shared pointer parameter is passed by reference and not reset or reassigned. Use T\* or T& instead."

**C++ Core Guidelines:** R.35: Take a shared\_ptr<widget>& parameter to express that a function might reseat the shared pointer

Passing shared pointers by reference may be useful in scenarios where callee code updates target of the smart pointer object and its caller expects to see such update. Using a reference solely to reduce costs of passing a shared pointer is questionable. If callee code only accesses target object and never manages its lifetime, it is safer to pass raw pointer or reference, rather than to expose resource management details.

## Remarks

- This check recognizes std::shared\_pointer and user-defined types which are likely to behave like shared pointers.  
The following traits are expected for user-defined shared pointers:
  - overloaded dereference or member access operators (public and non-deleted);
  - copy constructor or copy assignment operator (public and non-deleted);
  - public destructor which is neither deleted nor defaulted. Empty destructors are still counted as user-defined.
- The action of resetting or reassigning is interpreted in a more generic way:
- any call to a non-constant function on a shared pointer can potentially reset the pointer;
- any call to a function which accepts a reference to a non-constant shared pointer can potentially reset or reassign that pointer.

## Example

unnecessary interface complication

```
bool unregister(std::shared_ptr<event> &e) // C26417, also C26415 SMART_PTR_NOT_NEEDED
{
    return e && events_.erase(e->id());
}

void renew(std::shared_ptr<event> &e)
{
    if (unregister(e))
        e = std::make_shared<event>(e->id());
    // ...
}
```

## Example

unnecessary interface complication - simplified

```
bool unregister(const event *e)
{
    return e && events_.erase(e->id());
}

void renew(std::shared_ptr<event> &e)
{
    if (unregister(e.get()))
        e = std::make_shared<event>(e->id());
    // ...
}
```

# C26418 NO\_VALUE\_OR\_CONST\_REF\_SHARED\_PTR

3/21/2019 • 2 minutes to read • [Edit Online](#)

"Shared pointer parameter is not copied or moved. Use T\* or T& instead."

**C++ Core Guidelines:** R.36: Take a const shared\_ptr<widget>& parameter to express that it might retain a reference count to the object

If shared pointer parameter is passed by value or reference to a constant object it is expected that function will take control of its target object's lifetime without affecting of the caller. The code should either copy or move the shared pointer parameter to another shared pointer object or pass it further to other code by invoking functions which accept shared pointers. If this is not the case, then plain pointer or reference may be feasible.

## Remarks

- This check recognizes std::shared\_pointer and user-defined types which are likely to behave like shared pointers.  
The following traits are expected for user-defined shared pointers:
  - overloaded dereference or member access operators (public and non-deleted);
  - copy constructor or copy assignment operator (public and non-deleted);
  - public destructor which is neither deleted nor defaulted. Empty destructors are still counted as user-defined.

## Example

unnecessary interface complication

```
template<class T>
std::string to_string(const std::shared_ptr<T> &e) // C26418, also C26415 SMART_PTR_NOT_NEEDED
{
    return !e ? null_string : e->to_string();
}
```

## Example

unnecessary interface complication - simplified

```
template<class T>
std::string to_string(const T *e)
{
    return !e ? null_string : e->to_string();
}
```

# C26426 NO\_GLOBAL\_INIT\_CALLS

2/8/2019 • 2 minutes to read • [Edit Online](#)

"Global initializer calls a non-constexpr function."

**C++ Core Guidelines:** I.22: Avoid complex initialization of global objects

The order of execution of initializers for global objects may be inconsistent or undefined. This can lead to issues which are hard to reproduce and investigate. To avoid such problems global initializers should not depend on external code which is executed at runtime and can potentially depend on data which is not yet initialized. This rule flags cases where global objects use function calls to obtain their initial values.

## Remarks

- The rule ignores calls to constexpr functions or intrinsic functions on assumption that these either will be calculated at compile time or guarantee predictable execution at runtime.
- Calls to inline functions are still flagged since the checker doesn't attempt to analyze their implementation.
- This rule can be quite noisy in many common scenarios where a variable of a user defined type (or standard container) is initialized globally: this is often due to calls to constructors and destructors. This is still a valid warning since it points to places where unpredictable behavior may exist or future changes in external code may introduce instability.
- Static class members are considered global, so their initializers are also checked.

## Example

external version check

```
// api.cpp
int api_version = API_DEFAULT_VERSION; // Assume it can change at runtime, hence non-const.
int get_api_version() noexcept {
    return api_version;
}

// client.cpp
int get_api_version() noexcept;
bool is_legacy_mode = get_api_version() <= API_LEGACY_VERSION; // C26426, also stale value
```

## Example

external version check – made more reliable

```
// api.cpp
int& api_version() noexcept {
    static auto value = API_DEFAULT_VERSION;
    return value;
}
int get_api_version() noexcept {
    return api_version();
}

// client.cpp
int get_api_version() noexcept;
bool is_legacy_mode() noexcept {
    return get_api_version() <= API_LEGACY_VERSION;
}
```

# C26427 NO\_GLOBAL\_INIT\_EXTERNS

2/27/2019 • 2 minutes to read • [Edit Online](#)

"Global initializer accesses extern object."

**C++ Core Guidelines:** I.22: Avoid complex initialization of global objects

Global objects may be initialized in an inconsistent or undefined order which means that interdependency between them is risky and should be avoided. This is specifically applicable when initializers refer to another object considered to be 'extern'.

## Remarks

- An object is deemed extern if it conforms to the following rules:
  - it is a global variable marked with 'extern' specifier or it is a static member of a class;
  - it is not in an anonymous namespace;
  - it is not marked as 'const';
  - Static class members are considered global, so their initializers are also checked.

## Example

external version check // api.cpp

```
int api_version = API_DEFAULT_VERSION; // Assume it can change at runtime, hence non-const.

// client.cpp
extern int api_version;
bool is_legacy_mode = api_version <= API_LEGACY_VERSION; // C26427, also stale value
```

external version check – made more reliable

```
// api.cpp
int api_version = API_DEFAULT_VERSION; // Assume it can change at runtime, hence non-const.

// client.cpp
extern int api_version;
bool is_legacy_mode() noexcept
{
    return api_version <= API_LEGACY_VERSION;
}
```

# C26429 USE\_NONNULL

3/21/2019 • 2 minutes to read • [Edit Online](#)

"Symbol is never tested for nullness, it can be marked as `gsl::not_null`."

**C++ Core Guidelines:** F.23: Use a `not_null<T>` to indicate that "null" is not a valid value

It is a common practice to use asserts to enforce assumptions about validity of pointer values. The problem with asserts is that they do not expose assumptions through the interface (e.g. in return types or parameters). Asserts are also harder to maintain and keep in sync with other code changes. The recommendation is to use `gsl::not_null` from the Guidelines Support Library as a marker of resources which should never have null value. The rule `USE_NONNULL` helps to identify places that omit checks for nullness and hence can be updated to use `gsl::not_null`.

## Remarks

- The logic of the rule requires code to dereference a pointer variable so that nullness check (or enforcement of non-null value) would be justified. So, warning will be emitted only if pointers are dereferenced and never tested for nullness.
  - Current implementation handles only plain pointers (or their aliases) and doesn't detect smart pointers even though `gsl::not_null` can be applied to smart pointers as well.
  - A variable is marked as checked for nullness when it is used in the following contexts:
    - as a symbol expression in a branch condition, e.g. "if (p) { ... }";
    - non-bitwise logical operations;
    - comparison operations where one operand is a constant expression which evaluates to zero.
  - The rule doesn't have full dataflow tracking and can produce incorrect results in cases where indirect checks are used (e.g. when intermediate variable holds null value and later used in comparison).

## Example

hidden expectation

```
using client_collection = gsl::span<client*>;
// ...
void keep_alive(const connection *connection)    // C26429
{
    const client_collection clients = connection->get_clients();
    for (ptrdiff_t i = 0; i < clients.size(); i++)
    {
        auto client = clients[i];                // C26429
        client->send_heartbeat();
        // ...
    }
}
```

hidden expectation – clarified by `gsl::not_null`

```
using client_collection = gsl::span<gsl::not_null<client*>>;
// ...
void keep_alive(gsl::not_null<const connection*> connection)
{
    const client_collection clients = connection->get_clients();
    for (ptrdiff_t i = 0; i < clients.size(); i++)
    {
        auto client = clients[i];
        client->send_heartbeat();
        // ...
    }
}
```

# C26430 TEST\_ON\_ALL\_PATHS

3/21/2019 • 2 minutes to read • [Edit Online](#)

"Symbol is not tested for nullness on all paths."

**C++ Core Guidelines:** F.23: Use a `not_null<T>` to indicate that "null" is not a valid value

If code ever checks nullness of pointer variables it should do this consistently and validate pointers on all paths. Sometimes overaggressive checking for nullness is still better than possibility of a hard crash in one of the complicated branches. Ideally such code should be refactored to be less complex (by splitting into multiple functions) and to rely on markers like `gsl::not_null` (see Guidelines Support Library) to isolate parts of algorithm that can make safe assumption about valid pointer values. The rule `TEST_ON_ALL_PATHS` helps to find places where nullness checks are either inconsistent (hence assumptions may require review) or actual bugs where potential null value can bypass nullness check in some of the code paths.

## Remarks

- This rule expects that code dereferences a pointer variable so that nullness check (or enforcement of non-null value) would be justified. If there is no dereference, the rule is suspended.
  - Current implementation handles only plain pointers (or their aliases) and doesn't detect smart pointers even though nullness checks are applicable to smart pointers as well.
  - A variable is marked as checked for nullness when it is used in the following contexts:
    - as a symbol expression in a branch condition, e.g. "if (p) { ... }";
    - non-bitwise logical operations;
    - comparison operations where one operand is a constant expression which evaluates to zero.
  - The rule doesn't have full data flow tracking and can produce incorrect results in cases where indirect checks are used (e.g. when intermediate variable holds null value and later used in comparison).
  - Implicit nullness checks are assumed when pointer value is assigned from:
    - an allocation performed with throwing operator `new`;
    - a pointer obtained from type marked with `gsl::not_null`.

## Example

inconsistent testing reveals logic error

```
void merge_states(const state *left, const state *right) // C26430
{
    if (*left && *right)
        converge(left, right);
    else
    {
        // ...
        if (!left && !right)           // Logic error!
            discard(left, right);
    }
}
```

inconsistent testing reveals logic error - corrected

```
void merge_states(gsl::not_null<const state *> left, gsl::not_null<const state *> right)
{
    if (*left && *right)
        converge(left, right);
    else
    {
        // ...
        if (*left && *right)
            discard(left, right);
    }
}
```

# C26431 DONT\_TEST\_NOTNULL

3/21/2019 • 2 minutes to read • [Edit Online](#)

"The type of expression is already `gsl::not_null`. Do not test it for nullness."

**C++ Core Guidelines:** F.23: Use a `not_null<T>` to indicate that "null" is not a valid value

The marker type `gsl::not_null` from Guidelines Support Library is used to clearly indicate values which are never null pointers. It causes a hard failure if such assumption is not held at runtime. So, obviously, there is no need to check for nullness if expression evaluates to a result of type `gsl::not_null`.

## Remarks

- Since `gsl::not_null` itself is a thin pointer wrapper class, the rule actually tracks temporary variables that hold results from calls to the overloaded conversion operator (which returns contained pointer object). Such logic makes this rule applicable to expressions that involve variables and eventually have result of the `gsl::not_null` type. But it currently skips expressions that contain function calls returning `gsl::not_null`.
  - Current heuristic for nullness checks detects the following contexts:
    - symbol expression in a branch condition, e.g. "if (p) { ... }";
    - non-bitwise logical operations;
    - comparison operations where one operand is a constant expression which evaluates to zero.

## Example

unnecessary null checks reveal questionable logic

```
class type {
public:
    template<class T> bool is() const;
    template<class T> gsl::not_null<const T*> as() const;
    //...
};

class alias_type : public type {
public:
    gsl::not_null<const type*> get_underlying_type() const;
    gsl::not_null<const type*> get_root_type() const
    {
        const auto ut = get_underlying_type();
        if (ut)                                // C26431
        {
            const auto uat = ut->as<alias_type>();
            if (uat)                            // C26431, also incorrect use of API!
                return uat->get_root_type();

            return ut;
        }
        return this;                          // Alias to nothing? Actually, dead code!
    }
    //...
};
```

unnecessary null checks reveal questionable logic - reworked

```
//...
gsl::not_null<const type*> get_root_type() const
{
    const auto ut = get_underlying_type();
    if (ut->is<alias_type>())
        return ut->as<alias_type>()->get_root_type();

    return ut;
}
//...
```

# C26432 DEFINE\_OR\_DELETE\_SPECIAL\_OPS

2/8/2019 • 2 minutes to read • [Edit Online](#)

"If you define or delete any default operation in the type, define or delete them all."

**C++ Core Guidelines:** C.21: If you define or =delete any default operation, define or =delete them all

Special operations like constructors are assumed to alter behavior of types so that they rely more on language mechanisms to automatically enforce specific scenarios (the canonical example is resource management). If any of these operations is explicitly defined, defaulted or deleted (as an indication that user wants to avoid any special handling of a type) it would be inconsistent to leave the other operations from the same group unspecified (i.e. implicitly defined by compiler).

## Remarks

- This check implements "the rule of five" which treats the following operations as special:
  - copy constructors;
  - move constructors;
  - copy assignment operators;
  - move assignment operators;
  - destructors;
- The rule doesn't check if operations are defined in the same way, i.e. it is okay to mix deleted and defaulted operations with explicitly defined, but they all must be specified somehow if any of them appears.
- Access levels are not important and can also be mixed.
- The warning flags the first non-static function definition of a type, once per type.

# C26433 OVERRIDE\_EXPLICITLY

2/8/2019 • 2 minutes to read • [Edit Online](#)

Function should be marked with `override`

## C++ Core Guidelines

### C.128: Virtual functions should specify exactly one of virtual, override, or final

It is not required by compiler to clearly state that a virtual function overrides its base. Not specifying 'override' can cause subtle issues during maintenance if the virtual specification ever changes in the class hierarchy. This also decreases readability and makes interface's polymorphic behavior less obvious. If function is clearly marked as 'override', it enables compiler to check consistency of the interface and help to spot issues before they manifest themselves at runtime.

## Notes

1. This rule is not applicable to destructors. Destructors have their own specifics regarding virtuality.
2. The rule doesn't flag functions explicitly marked as 'final', which is itself a special kind of virtual specifier.
3. Warnings show up on function definitions, not declarations. This may be confusing since definitions do not have virtual specifiers, but the warning is still legit.

## Example: Implicit overriding

```
class Shape {  
public:  
    virtual void Draw() = 0;  
    // ...  
};  
  
class Ellipse : public Shape {  
public:  
    void Draw() { // C26433  
        //...  
    }  
};
```

## See Also

### C.128: Virtual functions should specify exactly one of virtual, override, or final

# C26434 DONT\_HIDE\_METHODS

2/8/2019 • 2 minutes to read • [Edit Online](#)

"Function hides a non-virtual function."

## C++ Core Guidelines

### [C.128: Virtual functions should specify exactly one of virtual, override, or final](#)

Introducing a function which has the same name as a non-virtual function in a base class is like introducing a variable name which conflicts with a name from outer scope. Furthermore, if signatures of functions mismatch, the intended overriding may turn into overloading. Overall, name hiding is dangerous and error-prone.

## Remarks

- Only non-overriding functions in current class are checked.
- Only non-virtual functions of base classes are considered.
- No signature matching is performed. Warnings are emitted if unqualified names match.

## See Also

### [C.128: Virtual functions should specify exactly one of virtual, override, or final](#)

# C26435 SINGLE\_VIRTUAL\_SPECIFICATION

2/8/2019 • 2 minutes to read • [Edit Online](#)

"Function should specify exactly one of 'virtual', 'override', or 'final'."

## C++ Core Guidelines

### C.128: Virtual functions should specify exactly one of virtual, override, or final

To improve readability the kind of virtual behavior should be stated clearly and without unnecessary redundancy. Even though virtual specifiers can be used simultaneously, it is better to specify one at a time to emphasize the most important aspect of virtual behavior. The following order of importance is apparent:

- plain virtual function;
- virtual function which explicitly overrides its base;
- virtual function which overrides its base and provides the final implementation in current inheritance chain.

## Notes

- This rule skips destructors since they have special rules regarding virtuality.
- Warnings show up on function definitions, not declarations. This may be confusing since definitions do not have virtual specifiers, but the warning is still legitimate.

## Example: Redundant specifier

```
class Ellipse : public Shape {
public:
    void Draw() override {
        //...
    }
};

class Circle : public Ellipse {
public:
    void Draw() override final { // C26435, only 'final' is necessary.
        //...
    }
};
```

## See Also

### C.128: Virtual functions should specify exactly one of virtual, override, or final

# C26436 NEED\_VIRTUAL\_DTOR

2/8/2019 • 2 minutes to read • [Edit Online](#)

"The type with a virtual function needs either public virtual or protected nonvirtual destructor."

**C++ Core Guidelines:** C.35: A base class destructor should be either public and virtual, or protected and nonvirtual

If a class defines a virtual function it becomes polymorphic, which implies that derived classes can change its behavior including resource management and destruction logic. Because client code may call polymorphic types via pointers to base classes, there is no way a client can explicitly choose which behavior is appropriate without downcasting. To make sure that resources are managed consistently and destruction occurs according to the actual type's rules it is recommended to define a public virtual destructor. If the type hierarchy is designed to disallow client code to destroy objects directly, destructors should be defined as protected non-virtual.

## Remarks

- The warning shows up on the first virtual function definition of a type (it can be a virtual destructor if it is not public), once per type.
  - Since definition can be placed separately from declaration, it may not always have any of the virtual specifiers. But the warning is still valid – it checks the actual 'virtuality' of a function.

# C26437 DONT\_SLICE

2/27/2019 • 2 minutes to read • [Edit Online](#)

"Do not slice."

**C++ Core Guidelines:** ES.63: Don't slice

Slicing is allowed by compiler and can be viewed as a special case of dangerous implicit cast. Even if it is done intentionally and doesn't lead to immediate issues, it is still highly discouraged since it makes code rather unmaintainable by forcing additional requirements on related data types. This is especially true if types are polymorphic or involve resource management.

## Remarks

- This rule would warn not only on explicit assignments, but also on implicit slicing which happens when result gets returned from current function or data passed as arguments to other functions.
  - Warnings would also flag cases where assignment doesn't involve real data slicing (e.g. if types are empty or don't make any dangerous data manipulations). Such warnings should still be addressed to prevent any undesirable regressions if types data or behavior changes in future.

## Example

slicing points to outdated

```
interface
struct id {
    int value;
};

struct id_ex : id {
    int extension;
};

bool read_id(stream &s, id &v) {
    id_ex tmp{};
    if (!s.read(tmp.value) || !s.read(tmp.extension))
        return false;

    v = tmp; // C26437
    return true;
}
```

slicing points to outdated, interface - corrected

```
// ...
bool read_id(stream &s, id_ex &v) {
// ...
```

# C26438 NO\_GOTO

2/27/2019 • 2 minutes to read • [Edit Online](#)

"Avoid 'goto'."

**C++ Core Guidelines:** ES.76: Avoid goto

Using of 'goto' is widely acknowledged as dangerous and error-prone practice. It is acceptable only in generated code (e.g. in a parser generated from a grammar). With modern C++ features and utilities provided by the Guidelines Support Library it should be easy to avoid 'goto' altogether.

## Remarks

- This rule warns on any occurrence of 'goto', even if it happens in dead code, except template code which is never used and hence ignored by compiler.
  - Warnings can be noisy if they encounter a macro containing 'goto'. Current reporting mechanism would point to all instances where such macro gets expanded. But the fix can usually be done in one place by changing the macro or avoiding use of it and leveraging more maintainable mechanisms.

## Example

'goto cleanup' in macro

```
#define ENSURE(E, L) if (!(E)) goto L;

void poll(connection &c)
{
    ENSURE(c.open(), end);           // C26438

    while (c.wait())
    {
        connection::header h{};
        connection::signature s{};
        ENSURE(c.read_header(h), end); // C26438
        ENSURE(c.read_signature(s), end); // C26438
        // ...
    }

    end:
    c.close();
}
```

'goto cleanup' in macro - replaced with gsl::finally

```
void poll(connection &c)
{
    auto end = gsl::finally([&c] { c.close(); });

    if (!c.open())
        return;

    while (c.wait())
    {
        connection::header h{};
        connection::signature s{};
        if(!c.read_header(h))
            return;
        if(!c.read_signature(s))
            return;
        // ...
    }
}
```

# C26439 SPECIAL\_NOEXCEPT

2/8/2019 • 2 minutes to read • [Edit Online](#)

"This kind of function may not throw. Declare it 'noexcept'."

**C++ Core Guidelines:** F.6: If your function may not throw, declare it noexcept

Some kinds of operations should never cause exceptions. Their implementations should be reliable and should handle possible errors conditions gracefully. They should never use exceptions to indicate failure. This rule flags cases where such operations are not explicitly marked as 'noexcept' which means that they may throw exceptions and cannot convey assumptions about their reliability.

## Remarks

- Special kinds of operations are the following:
  - destructors;
  - default constructors;
  - move constructors and move assignment operators;
  - standard functions with move semantics: std::move and std::swap.
  - Non-standard and outdated specifiers like throw() or declspec(nothrow) are not equivalent to 'noexcept'.
  - Explicit specifiers noexcept(false) and noexcept(true) are respected appropriately.
  - The warning may still appear for operations that are marked as constexpr. This may change in future releases.

# C26440 DECLARE\_NOEXCEPT

2/8/2019 • 2 minutes to read • [Edit Online](#)

"Function can be declared 'noexcept'."

**C++ Core Guidelines:** F.6: If your function may not throw, declare it noexcept

If code is not supposed to cause any exceptions, it should be marked as such by using the 'noexcept' specifier. This would help to simplify error handling on the client code side, as well as enable compiler to do additional optimizations.

## Remarks

- A function is considered non-throwing if:
  - it has no explicit throw statements;
  - function calls in its body, if any, invoke only functions that unlikely to throw: constexpr or functions marked with any exception specification which entails non-throwing behavior (this includes some non-standard specifications).
  - Non-standard and outdated specifiers like throw() or declspec(nothrow) are not equivalent to 'noexcept'.
  - Explicit specifiers noexcept(false) and noexcept(true) are respected appropriately.
  - Functions marked as constexpr are not supposed to cause exceptions and are not analyzed.
  - The rule also applies to lambda expressions.
  - The logic doesn't consider recursive calls as potentially non-throwing. This may change in the future.

# C26441 NO\_UNNAMED\_GUARDS

2/27/2019 • 2 minutes to read • [Edit Online](#)

"Guard objects must be named."

**C++ Core Guidelines:** CP.44: Remember to name your lock\_guards and unique\_locks

The standard library provides a few useful classes which help to control concurrent access to resources. Objects of such types lock exclusive access for the duration of their lifetime. This implies that every lock object must be named, i.e. have clearly defined lifetime which spans through the period in which access operations are executed. So, failing to assign a lock object to a variable is a mistake which is effectively disables locking mechanism (because temporary variables are transient). This rule tries to catch simple cases of such unintended behavior.

## Remarks

- Only standard lock types are tracked: std::scoped\_lock, std::unique\_lock, and std::lock\_guard.
  - Only simple calls to constructors are analyzed. More complex initializer expression may lead to inaccurate results, but this is rather an unusual scenario.
  - Locks passed as arguments to function calls or returned as results of function calls are ignored.
  - Locks created as temporaries but assigned to named references to extend their lifetime are ignored.

## Example

missing scoped variable

```
void print_diagnostic(gsl::string_span<> text)
{
    auto stream = get_diagnostic_stream();
    if (stream)
    {
        std::lock_guard<std::mutex>{ diagnostic_mutex_ }; // C26441
        write_line(stream, text);
        // ...
    }
}
```

missing scoped variable - corrected

```
void print_diagnostic(gsl::string_span<> text)
{
    auto stream = get_diagnostic_stream();
    if (stream)
    {
        std::lock_guard<std::mutex> lock{ diagnostic_mutex_ };
        write_line(stream, text);
        // ...
    }
}
```

# C26443 NO\_EXPLICIT\_DTOR\_OVERRIDE

2/8/2019 • 2 minutes to read • [Edit Online](#)

"Overriding destructor should not use explicit 'override' or 'virtual' specifiers."

## C++ Core Guidelines:

[C.128: Virtual functions should specify exactly one of virtual, override, or final.](#)

Destructors are generally very specific functions. This rule may be debatable but current consensus on the Core Guidelines is to exclude destructors from the 'override explicitly' recommendation.

## Notes

- The rule flags overriding destructors that explicitly use 'virtual' or 'override' specifiers.
- Destructors can still use the 'final' specifier due to its special semantics.
- Warnings show up on function definitions, not declarations. This may be confusing since definitions do not have virtual specifiers, but the warning is still legit.

## Example: Explicit 'override'

```
class Transaction {
public:
    virtual ~Transaction();
    // ...
};

class DistributedTransaction : public Transaction {
public:
    ~DistributedTransaction() override { // C26443
        // ...
    }
};
```

## See Also

[C.128: Virtual functions should specify exactly one of virtual, override, or final](#)

# C26444 NO\_UNNAMED\_RAIIL\_OBJECTS

2/8/2019 • 2 minutes to read • [Edit Online](#)

Avoid unnamed objects with custom construction and destruction.

## C++ Core Guidelines

### [ES.84: Don't \(try to\) declare a local variable with no name](#)

Unnamed (that is, temporary) objects with non-trivial behavior may point to either (a) inefficient code that allocates and immediately throws away resources or (b) to the code that unintentionally ignores non-primitive data. Sometimes it may also indicate plainly wrong declaration.

## Notes

- This rule detects types with non-deleted destructors. Keep in mind that destructors can be compiler generated.
- The warning can flag code that is not compiler generated and that invokes either a non-default constructor of a RAII type or a function that returns an object of such type. This warning helps to detect ignored call results in addition to wrong declarations.
- The logic skips temporaries if they are used in higher-level expressions. One example is temporaries that are passed as arguments or used to invoke a function.
- The standard library implementation may have different versions of destruction logic for some types (for example, containers). This can produce noisy warnings on debug builds because it is customary to ignore iterators returned from calls like `std::vector::insert`. While such warnings are not actionable in the majority of cases, they are legitimate in pointing to the place where some non-obvious work is done in temporary objects.

## Example: Ignored call result

```
std::string ToTraceMessage(State &state);
void SaveState(State &state)
{
    // ...
    ToTraceMessage(state); // C26444, should we do something with the result of this call?
}

Example: Ignored call result - fixed.
std::cerr << ToTraceMessage(state);

Example: Unexpected lifetime.
void SplitCache()
{
    gsl::finally([] { RestoreCache(); }); // C26444, RestoreCache is invoked immediately!
    //...
}

Example: Unexpected lifetime - fixed.
const auto _ = gsl::finally([] { RestoreCache(); });
```

## See Also

### [ES.84: Don't \(try to\) declare a local variable with no name](#)

# C26445 NO\_SPAN\_REF

2/8/2019 • 2 minutes to read • [Edit Online](#)

A reference to `gsl::span` or `std::string_view` may be an indication of a lifetime issue. C++ Core Guidelines: [GSL.view: Views](#)

The intention of this rule is to catch subtle lifetime issues that may occur in code which has been migrated from standard containers to new span and view types. Such types can be considered as “references to buffers.” Using a reference to a span or view creates an additional layer of indirection. Such indirection is often unnecessary and can be confusing for maintainers. In addition, spans are cheap to copy and can be returned by value from function calls. Obviously, such call results should never be referenced.

## Remarks

- The rule detects references to `gsl::span<>`, `gsl::basic_string_span<>`, and `std::basic_string_view<>` (including aliases to instantiations).
- Currently warnings are emitted only on declarations and return statements. This may be extended in future to also flag function parameters.
- The implementation of this rule is very lightweight doesn’t attempt to trace actual lifetimes. Using of references may still make sense in some scenarios. In such cases, false positives can safely be suppressed.

## Example: Reference to a temporary

```
// Old API - uses string reference to avoid data copy.  
const std::string& get_working_directory() noexcept;  
  
// New API - after migration to C++17 it uses string view.  
std::string_view get_working_directory() noexcept;  
  
// ...  
// Client code which places an explicit reference in a declaration with auto specifier.  
const auto &wd = get_working_directory(); // C26445 after API update.
```

# C26446 USE\_GSL\_AT

4/16/2019 • 2 minutes to read • [Edit Online](#)

Prefer to use `gsl::at()` instead of unchecked subscript operator.

C++ Core Guidelines: [Bounds.4: Don't use standard-library functions and types that are not bounds-checked](#).

The Bounds profile of the C++ Core Guidelines tries to eliminate unsafe manipulations of memory by avoiding the use of raw pointers and unchecked operations. One way to perform uniform range-checked access to buffers is to use the `gsl::at()` utility from the Guidelines Support Library. It is also a good practice to rely on standard implementations of `at()` available in STL containers.

This rule helps to find places where potentially unchecked access is performed via calls to `operator[]()`. In most cases such calls can be easily replaced by `gsl::at()`.

- Access to arrays with known size is flagged when non-constant index is used in a subscript operator. Constant indices are handled by [C26483 STATIC\\_INDEX\\_OUT\\_OF\\_RANGE](#).
- The logic to warn on overloaded `operator[]` calls is more complex:
  - If index is non-integral, the call is ignored. This also handles indexing in standard maps since parameters in such operators are passed by reference.
  - If the operator is marked as non-throwing (by using `noexcept`, `throw()`, or `__declspec(nothrow)`), the call is flagged. It is assumed that if the subscript operator never throws exceptions it either doesn't perform range checks or these checks are obscure.
  - If the operator is not marked as non-throwing, it may be flagged if it comes from an STL container that also defines a conventional `at()` member function (such functions are detected by simple name matching).
  - The rule doesn't warn on calls to standard `at()` functions. These functions are safe and replacing them with `gsl::at()` would not bring much value.
- Indexing into `std::basic_string_view<>` is not safe, so a warning is issued. The standard string view can be replaced by `gsl::basic_string_span<>`, which is always bounds-checked.
- The implementation doesn't consider range checks that user code may have somewhere in loops or branches. Accuracy here is traded for performance. In general, explicit range checks can often be replaced with more reliable iterators or more concise enhanced for-loops.

# C26447 DONT\_THROW\_IN\_NOEXCEPT

2/8/2019 • 2 minutes to read • [Edit Online](#)

The function is declared **noexcept** but calls a function that may throw exceptions.

C++ Core Guidelines: [F.6: If your function may not throw, declare it noexcept](#)

This rule amends another rule, [C26440 DECLARE\\_NOEXCEPT](#), which tries to find functions that are good candidates to be marked as **noexcept**. In this case, the idea is that once some function is marked as **noexcept**, it must keep its contract by not invoking other code that may throw exceptions.

- The Microsoft C++ compiler already handles straightforward violations like **throw** statements in the function body (see [C4297](#)).
- The rule focuses only on function calls. It flags targets that are not **constexpr** and that can potentially throw exceptions; in other words they are not marked explicitly as non-throwing by using **noexcept**, **\_declspec(nothrow)**, **throw()**.
- The compiler-generated target functions are skipped to reduce noise since exception specifications are not always provided by the compiler.
- The checker also skips special kinds of target functions that are expected to be implemented as **noexcept**; this rule is enforced by [C26439 SPECIAL\\_NOEXCEPT](#).

# C26448 USE\_GSL\_FINALLY

2/8/2019 • 2 minutes to read • [Edit Online](#)

Consider using `gsl::finally` if final action is intended.

C++ Core Guidelines: [GSL.util: Utilities](#)

The Guidelines Support Library provides a convenient utility to implement the *final action* concept. Since the C++ language doesn't support **try-finally** constructs, it became common to implement custom cleanup types that would invoke arbitrary actions on destruction. The `gsl::finally` utility is implemented in this way and provides a more uniform way to perform final actions across a code base.

There are also cases where final actions are performed in an old-fashioned C-style way by using **goto** statements (which is generally discouraged by [C26438 NO\\_GOTO](#)). It is hard to detect the exact intention in code that heavily uses **goto**, but some heuristics can help to find better candidates for cleanup.

## Remarks

- This rule is very lightweight and uses label names to guess about opportunities to use final action objects.
- Label names that can raise a warning contain words like "end", "final", "clean", and so on.
- Warnings appear at the **goto** statements. You may see verbose output on some occasions, but this may help in prioritizing code depending on its complexity.
- This rule always goes in pair with [C26438 NO\\_GOTO](#). Depending on the priorities, one of these can be disabled.

## Example: Cleanup with multiple goto statements

```
void poll(connection_info info)
{
    connection c = {};
    if (!c.open(info))
        return;

    while (c.wait())
    {
        connection::header h{};
        connection::signature s{};
        if (!c.read_header(h))
            goto end;           // C26448 and C26438
        if (!c.read_signature(s))
            goto end;           // C26448 and C26438
        // ...
    }

end:
    c.close();
}
```

## Example: Cleanup with multiple goto statements replaced by gsl::finally

```
void poll(connection_info info)
{
    connection c = {};
    if (!c.open(info))
        return;

    auto end = gsl::finally([&c] { c.close(); });

    while (c.wait())
    {
        connection::header h{};
        connection::signature s{};

        if (!c.read_header(h))
            return;
        if (!c.read_signature(s))
            return;
        // ...
    }
}
```

# C26449 NO\_SPAN\_FROM\_TEMPORARY

2/8/2019 • 2 minutes to read • [Edit Online](#)

`gsl::span` or `std::string_view` created from a temporary will be invalid when the temporary is invalidated.

C++ Core Guidelines: [GSL.view: Views](#).

Spans and views are convenient and lightweight types that allow to reference memory buffers. But they must be used carefully: while their interface looks similar to standard containers, their behavior is more like the behavior of pointers and references. They do not own data and must never be constructed from temporary buffers. This check focuses on cases where source data is temporary, while span or view is not. There is another check which handles slightly different scenario involving span references: [C26445 NO\\_SPAN\\_REF](#). Both rules can help to avoid subtle but dangerous mistakes made when legacy code gets modernized and adopts spans or views.

## Remarks

- This rule warns on places where constructors get invoked for spans or views and the source data buffer belongs to a temporary object created in the same statement. This includes:
  - implicit conversions in return statements;
  - implicit conversions in ternary operators;
  - explicit conversions in `static_cast` expressions;
  - function calls that return containers by value.
- Temporaries created for function call arguments are not flagged. It is safe to pass spans from such temporaries if target functions don't retain data pointers in external variables.
- If spans or views are themselves temporaries, the rule skips them.
- Data tracking in the checker has certain limitations; therefore complex scenarios involving multiple or obscure reassignments may not be handled.

## Example: Subtle difference in result types

```
// Returns a predefined collection. Keeps data alive.  
gsl::span<const sequence_item> get_seed_sequence() noexcept;  
  
// Returns a generated collection. Doesn't own new data.  
const std::vector<sequence_item> get_next_sequence(gsl::span<const sequence_item>);  
  
void run_batch()  
{  
    auto sequence = get_seed_sequence();  
    while (send(sequence))  
    {  
        sequence = get_next_sequence(sequence); // C26449  
        // ...  
    }  
}
```

# Arithmetic overflow: '%operator%' operation causes overflow at compile time. Use a wider type to store the operands

2/8/2019 • 2 minutes to read • [Edit Online](#)

This warning indicates that an arithmetic operation was provably lossy at compile time. This can be asserted when the operands are all compile-time constants. Currently, we check left shift, multiplication, addition, and subtraction operations for such overflows.

Note: C4307 is a similar check in the Microsoft C++ compiler.

## Example 1

```
int multiply()
{
    const int a = INT_MAX;
    const int b = 2;
    int c = a * b; // C26450 reported here
    return c;
}
```

To correct this warning, use the following code.

```
long long multiply()
{
    const int a = INT_MAX;
    const int b = 2;
    long long c = (long long)a * b; // OK
    return c;
}
```

## Example 2

```
int add()
{
    const int a = INT_MAX;
    const int b = 2;
    int c = a + b; // C26450 reported here
    return c;
}
```

To correct this warning, use the following code:

```
long long add()
{
    const int a = INT_MAX;
    const int b = 2;
    long long c = (long long)a + b; // OK
    return c;
}
```

## Example 3

```
int subtract()
{
    const int a = -INT_MAX;
    const int b = 2;
    int c = a - b; // C26450 reported here
    return c;
}
```

To correct this warning, use the following code.

```
long long subtract()
{
    const int a = -INT_MAX;
    const int b = 2;
    long long c = (long long)a - b; // OK
    return c;
}
```

## See Also

[ES.103: Don't overflow](#)

# Warning C26451: Arithmetic overflow: Using operator '%operator%' on a %size1% byte value and then casting the result to a %size2% byte value. Cast the value to the wider type before calling operator '%operator%' to avoid overflow

2/8/2019 • 2 minutes to read • [Edit Online](#)

This warning indicates incorrect behavior that results from integral promotion rules and types larger than those in which arithmetic is typically performed.

We detect when a narrow type integral value was shifted left, multiplied, added, or subtracted and the result of that arithmetic operation was cast to a wider type value. If the operation overflowed the narrow type value, then data is lost. You can prevent this loss by casting the value to a wider type before the arithmetic operation.

## Example 1

The following code generates this warning:

```
void leftshift(int i)
{
    unsigned __int64 x;
    x = i << 31; // C26451 reported here

    // code
}
```

To correct this warning, use the following code:

```
void leftshift(int i)
{
    unsigned __int64 x;
    x = (unsigned __int64)i << 31; // OK

    // code
}
```

## Example 2

```
void somefunc(__int64 y);

void callsomefunc(int x)
{
    somefunc(x * 2); // C26451 reported here
}
```

To correct this warning, use the following code:

```
void callsomefunc(int x)
{
    somefunc((__int64)x * 2); // OK
}
```

## Example 3

```
__int64 add(int x)
{
    constexpr auto value = 2;
    return x += value; // C26451 reported here
}
```

To correct this warning, use the following code:

```
__int64 add(int x)
{
    constexpr auto value = 2;
    const __int64 y = (__int64)x + value; // OK
    return y;
}
```

## See also

- [ES.103: Don't overflow](#)

# Arithmetic overflow: Left shift count is negative or greater than or equal to the operand size which is undefined behavior

2/8/2019 • 2 minutes to read • [Edit Online](#)

This warning indicates shift count is negative or greater than or equal to the number of bits of the operand being shifted, resulting in undefined behavior. Note: C4293 is a similar check in the Microsoft C++ compiler.

## Example

```
unsigned __int64 combine(unsigned lo, unsigned hi)
{
    return (hi << 32) | lo; // C26252 here
}
```

To correct this warning, use the following code:

```
unsigned __int64 combine(unsigned lo, unsigned hi)
{
    return ((unsigned __int64)hi << 32) | lo; // OK
}
```

## See Also

[ES.102: Use signed types for arithmetic](#)

# Warning C26253: Arithmetic overflow: Left shift of a negative signed number is undefined behavior

2/8/2019 • 2 minutes to read • [Edit Online](#)

This warning indicates we are left shifting a negative signed integral value, which is a bad idea and triggers implementation defined behavior.

## Example

```
void leftshift(int shiftCount)
{
    const auto result = -1 << shiftCount; // C26453 reported here

    // code
}
```

To correct this warning, use the following code:

```
void leftshift(int shiftCount)
{
    const auto result = ((unsigned)-1) << shiftCount; // OK

    // code
}
```

## See Also

[ES.102: Use signed types for arithmetic](#)

# Arithmetic overflow: '%operator%' operation produces a negative unsigned result at compile time

2/8/2019 • 2 minutes to read • [Edit Online](#)

This warning indicates that the subtraction operation produces a negative result which was evaluated in an unsigned context. This can result in unintended overflows.

## Example

```
unsigned int negativeunsigned()
{
    const unsigned int x = 1u - 2u; // C26454 reported here
    return x;
}
```

To correct this warning, use the following code:

```
unsigned int negativeunsigned()
{
    const unsigned int x = 4294967295; // OK
    return x;
}
```

## See Also

[ES.106: Don't try to avoid negative values by using unsigned](#)

# C26455 DEFAULTCTOR\_NOEXCEPT

2/8/2019 • 2 minutes to read • [Edit Online](#)

The C++ Core Guidelines suggest that default constructors shouldn't do anything that can throw. If the default constructor is allowed to throw, operations such as move and swap will also throw which is undesirable because move and swap should always succeed. Parameterized constructors may throw.

## Remarks

Consider the default constructors of the STL types, like `std::vector`. In these implementations, the default constructors initialize internal state without making allocations. In the `std::vector` case, the size is set to 0 and the internal pointer is set to `nullptr`. The same pattern should be followed for all default constructors.

## See also

- [C++ Core Guideline for this warning](#)

# C26456 DONT\_HIDE\_OPERATORS

2/8/2019 • 2 minutes to read • [Edit Online](#)

Hiding base methods that aren't virtual is error prone and makes code harder to read.

[C++ Core Guideline for this warning](#)

# C26460 USE\_CONST\_REFERENCE\_ARGUMENTS

2/8/2019 • 2 minutes to read • [Edit Online](#)

The reference argument '%argument%' for function '%function%' can be marked as `const`. See [C++ Core Guidelines con.3](#).

# C26461 USE\_CONST\_POINTER\_ARGUMENTS:

2/8/2019 • 2 minutes to read • [Edit Online](#)

The pointer argument '%argument%' for function '%function%' can be marked as a pointer to `const`. See [C++ Core Guidelines con.3](#).

# C26462 USE\_CONST\_POINTER\_FOR\_VARIABLE

2/8/2019 • 2 minutes to read • [Edit Online](#)

The value pointed to by '%variable%' is assigned only once, mark it as a pointer to `const`. See [C++ Core Guidelines con.4](#).

# C26463 USE\_CONST\_FOR\_ELEMENTS

2/8/2019 • 2 minutes to read • [Edit Online](#)

The elements of array '%array%' are assigned only once, mark elements `const`. See [C++ Core Guidelines con.4](#).

# C26464 USE\_CONST\_POINTER\_FOR\_ELEMENTS

2/8/2019 • 2 minutes to read • [Edit Online](#)

The values pointed to by elements of array '%array%' are assigned only once, mark elements as pointer to `const`.

See [C++ Core Guidelines con.4](#).

# C26465 NO\_CONST\_CAST\_UNNECESSARY

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't use `const_cast` to cast away `const`. `const_cast` is not required; constness or volatility is not being removed by this conversion. See [C++ Core Guidelines Type.3](#).

# C26466 NO\_STATIC\_DOWNCASE\_POLYMORPHIC

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't use `static_cast` downcasts. A cast from a polymorphic type should use `dynamic_cast`. See [C++ Core Guidelines Type.2](#).

# C26471 NO\_REINTERPRET\_CAST\_FROM\_VOID\_PTR

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't use `reinterpret_cast`. A cast from void\* can use `static_cast`. See [C++ Core Guidelines Type.1](#).

# C26472

## NO\_CASTS\_FOR\_ARITHMETIC\_CONVERSION

2/27/2019 • 2 minutes to read • [Edit Online](#)

"Don't use a static\_cast for arithmetic conversions. Use brace initialization, gsl::narrow or gsl::narrow."

**C++ Core Guidelines:** Type.1: Avoid casts

This rule helps to find places where static casts are used to convert between integral types, which is unsafe since compiler would not warn if any data loss occurs. Brace initializers are better for the cases where constants are used, and a compiler error is desired. There are also utilities from the Guidelines Support Library that help to describe intentions clearly:

- gsl::narrow ensures lossless conversion and causes runtime crash if it is not possible.
- gsl::narrow\_cast clearly states that conversion can lose data and it is acceptable.

### Remarks

- This rule is implemented only for static\_casts. Using of C-style casts is generally discouraged.

### Example

unhandled unexpected data

```
rgb from_24bit(std::uint32_t v) noexcept {
    return {
        static_cast<std::uint8_t>(v >> 16),           // C26472, what if top byte is non-zero?
        static_cast<std::uint8_t>((v >> 8) & 0xFF), // C26472
        static_cast<std::uint8_t>(v & 0xFF)           // C26472
    };
}
```

unhandled unexpected data – safer version

```
rgb from_24bit(std::uint32_t v) noexcept {
    return {
        gsl::narrow<std::uint8_t>(v >> 16),
        gsl::narrow_cast<std::uint8_t>((v >> 8) & 0xFF),
        gsl::narrow_cast<std::uint8_t>(v & 0xFF)
    };
}
```

# C26473 NO\_IDENTITY\_CAST

2/27/2019 • 2 minutes to read • [Edit Online](#)

"Don't cast between pointer types where the source type and the target type are the same."

**C++ Core Guidelines:** Type.1: Avoid casts

This rule helps to remove unnecessary or suspicious casts. Obviously, when type is converted to itself, such conversion is ineffective, yet the fact that the cast is used may indicate subtle design issue or a potential for regression if types change in future. It is always safer to use as few casts as possible.

## Remarks

- This rule is implemented for static and reinterpret casts and checks only pointer types.

## Example

dangerously generic lookup

```
gsl::span<server> servers_;
```

```
template<class T>
server* resolve_server(T tag) noexcept {
    auto p = reinterpret_cast<server*>(tag); // C26473, also 26490 NO_REINTERPRET_CAST
    return p >= &(*servers_.begin()) && p < &(*servers_.end()) ? p : nullptr;
}

void promote(server *s, int index) noexcept {
    auto s0 = resolve_server(s);
    auto s1 = resolve_server(index);
    if (s0 && s1)
        std::swap(s0, s1);
}
```

dangerously generic lookup - reworked

```
// ...
server* resolve_server(server *p) noexcept {
    return p >= &(*servers_.begin()) && p < &(*servers_.end()) ? p : nullptr;
}

server* resolve_server(ptrdiff_t i) noexcept {
    return !servers_.empty() && i >= 0 && i < servers_.size() ? &servers_[i] : nullptr;
}
// ...
```

# C26474 NO\_IMPLICIT\_CAST

2/27/2019 • 2 minutes to read • [Edit Online](#)

"Don't cast between pointer types when the conversion could be implicit."

**C++ Core Guidelines:** Type.1: Avoid casts

In some cases, implicit casts between pointer types can safely be done and don't require user to write specific cast expression. This rule finds instances of such unnecessary casting which can be removed.

## Remarks

- The rule ID is a bit misleading: it should be interpreted as "implicit cast is not used where it is acceptable".
  - The rule is applicable to pointers only and checks static casts and reinterpret casts.
  - The following cases are acceptable pointer conversions that should not use explicit cast expressions:
    - conversion to nullptr\_t;
    - conversion to void\*;
    - conversion from derived type to its base.

## Example

unnecessary conversion hides logic error

```
template<class T>
bool register_buffer(T buffer) {
    auto p = reinterpret_cast<void*>(buffer); // C26474, also 26490 NO_REINTERPRET_CAST
    return buffers_.insert(p).second;
}

void merge_bytes(std::uint8_t *left, std::uint8_t *right)
{
    if (left && register_buffer(*left)) { // Unintended dereference!
        // ...
        if (right && register_buffer(right)) {
            // ...
        }
    }
}
```

unnecessary conversion hides logic error - reworked

```
// ...
template<class T>
bool register_buffer(T *buffer) {
    auto p = buffer;
    return buffers_.insert(p).second;
}
// ...
```

# C26475 NO\_FUNCTION\_STYLE\_CASTS

2/27/2019 • 2 minutes to read • [Edit Online](#)

"Do not use function style C-casts."

**C++ Core Guidelines:** ES.49: If you must use a cast, use a named cast

Function style casts (e.g. "int(1.1)") are another incarnation of C-style casts (like "(int)1.1") with all its questionable safety. Specifically, compiler doesn't try to check if any data loss can occur neither in C-casts, nor in function casts. In both cases it is better either to avoid casting or use brace initializer if possible. If neither works, static casts may be suitable, but it is still better to use utilities from the Guidelines Support Library:

- `gsl::narrow` ensures lossless conversion and causes runtime crash if it is not possible.
- `gsl::narrow_cast` clearly states that conversion can lose data and it is acceptable.

## Remarks

- This rule fires only for constants of primitive types - these are the cases where compiler can clearly detect data loss and emit error if brace initializer is used. The cases which would require runtime execution are flagged by C26493 NO\_CSTYLE\_CAST.
- Default initializers are not flagged (e.g. "int()").

## Example

dangerous conversion

```
constexpr auto planck_constant = float( 6.62607004082e-34 ); // C26475
```

```
dangerous conversion - detecting potential data loss
constexpr auto planck_constant = float{ 6.62607004082e-34 }; // Error C2397
```

```
dangerous conversion - corrected
constexpr auto planck_constant = double{ 6.62607004082e-34 };
```

# C26476 USE\_VARIANT

2/8/2019 • 2 minutes to read • [Edit Online](#)

`std::variant` provides a type-safe alternative to `union` and should be preferred in modern code.

[C++ Core Guideline for this warning](#)

# C26477 USE\_NONNULLPTR\_NOT\_CONSTANT

2/8/2019 • 2 minutes to read • [Edit Online](#)

`nullptr` has a special type `nullptr_t` that allows overloads with special null handling. Using `0` or `NULL` in place of `nullptr` bypasses the type safety and deduction that `nullptr` provides.

[C++ Core Guideline for this warning](#)

# C26481 NO\_POINTER\_ARITHMETIC

2/8/2019 • 2 minutes to read • [Edit Online](#)

This check supports the rule *I.13: Do not pass an array as a single pointer*. Whenever raw pointers are used in arithmetic operations they should be replaced with safer kinds of buffers like `span<T>` or `vector<T>`.

## Remarks

- This check is a bit more restrictive than I.13: it doesn't skip zstring or czstring types.
- C26481 and C26485 come from the [Bounds Safety Profile](#) rules implemented in the first release of the C++ Core Guidelines Checker. They are applicable to raw pointers category since they help to avoid unsafe use of raw pointers.

# C26482 NO\_DYNAMIC\_ARRAY\_INDEXING

2/8/2019 • 2 minutes to read • [Edit Online](#)

Only index into arrays using constant expressions. See [C++ Core Guidelines Bounds.2](#)

# C26483 STATIC\_INDEX\_OUT\_OF\_RANGE

2/8/2019 • 2 minutes to read • [Edit Online](#)

Value %value% is outside the bounds (0, %bound%) of variable '%variable%'. Only index into arrays using constant expressions that are within bounds of the array. See [C++ Core Guidelines Bounds.2](#)

# C26485 NO\_ARRAY\_TO\_POINTER\_DECAY

2/8/2019 • 2 minutes to read • [Edit Online](#)

Like C26481, this check helps to enforce the rule I.13: *Do not pass an array as a single pointer* by detecting places where static array type information gets lost due to decay to a raw pointer. `zstring` and `czstring` types are not excluded.

## Remarks

C26481 and C26485 come from the [Bounds Safety Profile](#) rules implemented in the first release of the C++ Core Guidelines Checker. They are applicable to raw pointers category since they help to avoid unsafe use of raw pointers.

# C26486

## LIFETIMES\_FUNCTION\_PRECONDITION\_VIOLATION

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't pass a pointer that may be invalid (dangling) as a parameter to a function.

```
void use(int*);  
  
void ex1()  
{  
    int* px;  
    {  
        int x;  
        px = &x;  
    }  
  
    use(px); // px is a dangling pointer  
}
```

## Remarks

The Lifetime guidelines from the C++ core guidelines outline a contract that code can follow which will enable more thorough static memory leak and dangling pointer detection. The basic ideas behind the guidelines are:

- Never dereference an invalid (dangling) or known-null pointer
- Never return (either formal return or out parameter) any pointer from a function.
- Never pass an invalid (dangling) pointer to any function.

## See also

- [C++ Core Guidelines Lifetimes Paper](#)

# C26487

## LIFETIMES\_FUNCTION\_POSTCONDITION\_VIOLATION

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't allow a function to return an invalid pointer, either through the formal return or output parameters.

```
int* ex1(int a)
{
    return &a;      // returns a dangling pointer to the stack variable 'a'
}

void ex2(int a, int** out)
{
    *out = &a;    // 'out' contains a dangling pointer to the stack variable 'a'
}
```

### Remarks

The Lifetime guidelines from the C++ core guidelines outline a contract that code can follow which will enable more thorough static memory leak and dangling pointer detection. The basic ideas behind the guidelines are:

- Never dereference an invalid (dangling) or known-null pointer
- Never return (either formal return or out parameter) any pointer from a function.
- Never pass an invalid (dangling) pointer to any function.

### See also

- [C++ Core Guidelines Lifetimes Paper](#)

# C26488 LIFETIMES\_DEREF\_NULL\_POINTER

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't dereference a pointer that may be null.

```
void ex1()
{
    int* px = nullptr;

    if (px)      // notice the condition is incorrect
        return;

    *px = 1;     // 'px' known to be null here
}
```

## Remarks

The Lifetime guidelines from the C++ core guidelines outline a contract that code can follow which will enable more thorough static memory leak and dangling pointer detection. The basic ideas behind the guidelines are:

1. Never dereference an invalid (dangling) or known-null pointer
2. Never return (either formal return or out parameter) any pointer from a function.
3. Never pass an invalid (dangling) pointer to any function.

## See also

- [C++ Core Guidelines Lifetimes Paper](#)

# C26489 LIFETIMES\_DEREF\_INVALID\_POINTER

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't dereference a pointer that may be invalid.

```
int ex1()
{
    int* px;

    {
        int x = 0;
        px = &x;
    }

    return *px;    // 'px' was invalidated when 'x' went out of scope.
}
```

## Remarks

The Lifetime guidelines from the C++ core guidelines outline a contract that code can follow which will enable more thorough static memory leak and dangling pointer detection. The basic ideas behind the guidelines are:

1. Never dereference an invalid (dangling) or known-null pointer
2. Never return (either formal return or out parameter) any pointer from a function.
3. Never pass an invalid (dangling) pointer to any function.

## See also

[C++ Core Guidelines Lifetimes Paper](#)

# C26490 NO\_REINTERPRET\_CAST

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't use `reinterpret_cast`. See [C++ Core Guidelines Type.1](#).

# C26491 NO\_STATIC\_DOWNCASE

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't use `static_cast` downcasts. See [C++ Core Guidelines Type.2](#).

# C26492 NO\_CONST\_CAST

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't use `const_cast` to cast away `const`. See [C++ Core Guidelines Type.3](#).

# C26493 NO\_CSTYLE\_CAST

2/8/2019 • 2 minutes to read • [Edit Online](#)

Don't use C-style casts. See [C++ Core Guidelines Type.4](#).

# C26494 VAR\_USE\_BEFORE\_INIT

2/8/2019 • 2 minutes to read • [Edit Online](#)

Variable '%variable%' is uninitialized. Always initialize an object. See [C++ Core Guidelines Type.5](#).

# C26495 MEMBER\_UNINIT

2/8/2019 • 2 minutes to read • [Edit Online](#)

Variable '%variable%' is uninitialized. Always initialize a member variable. See [C++ Core Guidelines Type.6](#).

# C26496 USE\_CONST\_FOR\_VARIABLE

2/8/2019 • 2 minutes to read • [Edit Online](#)

The variable '%variable%' is assigned only once, mark it as `const`. See [C++ Core Guidelines con.4](#).

# C26497 USE\_CONSTEXPR\_FOR\_FUNCTION

2/8/2019 • 2 minutes to read • [Edit Online](#)

This function %function% could be marked `constexpr` if compile-time evaluation is desired. See [C++ Core Guidelines F.4](#).

# C26498 USE\_CONSTEXPR\_FOR\_FUNCTIONCALL

2/8/2019 • 2 minutes to read • [Edit Online](#)

This function call %function% can use `constexpr` if compile-time evaluation is desired. See [C++ Core Guidelines con.5](#).

# Code Analysis for C/C++ Warnings

2/8/2019 • 2 minutes to read • [Edit Online](#)

This section lists C/C++ Code Analysis warnings except those that are raised by the [C++ Core Guidelines Checkers](#). For information about Code Analysis, see [/analyze \(Code Analysis\)](#) and [Quick Start: Code Analysis for C/C++](#).

## See Also

- [Analyzing C/C++ Code Quality by Using Code Analysis](#)
- [Using SAL Annotations to Reduce C/C++ Code Defects](#)

# C1250

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C1250: Unable to load plug-in.

The Code Analysis tool reports this warning when there is an internal error in the plugin, not in the code being analyzed.

# C1251

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C1251: Unable to load models.

The Code Analysis tool reports this warning when there is an internal error in the model file, not in the code being analyzed.

# C1252

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C1252: Circular or missing dependency between plugins: requires GUID

The Code Analysis tool reports this warning when there is an internal error with plugin dependencies, not in the code being analyzed.

# C1253

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C1253: Unable to load model file.

The Code Analysis tool reports this warning when there is an internal error in the model file, not in the code being analyzed.

# C1254

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C1254: Plugin version mismatch : version doesn't match the version of the PREfast driver

The Code Analysis tool reports this warning when there is an internal error with the plugin version, not in the code being analyzed.

# C1255

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C1255: PCH data for plugin has incorrect length.

The Code Analysis tool reports this warning when there is an internal error in the tool, not in the code being analyzed.

# C1256

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C1256: PCH must be both written and read.

The Code Analysis tool reports this warning when there is an internal error in the tool, not in the code being analyzed.

# C1257

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 1257: Plugin Initialization Failure.

The Code Analysis tool reports this warning when there is an internal error in the plugin, not in the code being analyzed.

# C6001

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6001: using uninitialized memory <variable>

This warning is reported when an uninitialized local variable is used before it is assigned a value. This could lead to unpredictable results. You should always initialize variables before use.

## Example

The following code generates this warning because variable `i` is only initialized if `b` is true; otherwise an uninitialized `i` is returned:

```
int f( bool b )
{
    int i;
    if ( b )
    {
        i = 0;
    }
    return i; // i is uninitialized if b is false
}
```

To correct this warning, initialize the variable as shown in the following code:

```
int f( bool b )
{
    int i= -1;

    if ( b )
    {
        i = 0;
    }
    return i;
}
```

## See Also

[Compiler Warning \(level 1 and level 4\) C4700](#)

# C6011

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6011: dereferencing NULL pointer <name>

This warning indicates that a null pointer is being dereferenced. If the pointer value is invalid, the result is undefined.

## Example

The following code generates this warning because a call to malloc might return null if there is insufficient memory available:

```
#include <malloc.h>

void f( )
{
    char *p = ( char * ) malloc( 10 );
    *p = '\0';

    // code ...
    free( p );
}
```

To correct this warning, examine the pointer for null value as shown in the following code:

```
#include <malloc.h>
void f( )
{
    char *p = ( char * ) malloc ( 10 );
    if ( p )
    {
        *p = '\0';
        // code ...

        free( p );
    }
}
```

You must allocate memory inside the function whose parameters are annotated by using the Null property in a Pre condition before dereferencing the parameter. The following code generates warning C6011 because an attempt is made to dereference a null pointer (`pc`) inside the function without first allocating memory:

```
#include <sal.h>
using namespace vc_attributes;
void f([Pre(Null=Yes)] char* pc)
{
    *pc='\0'; // warning C6011 - pc is null
    // code ...
}
```

The use of malloc and free have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and

## See Also

- [Using SAL Annotations to reduce code defects](#)
- [NULL](#)
- [Indirection and Address-of Operators](#)
- [malloc](#)
- [free](#)

# C6014

2/21/2019 • 2 minutes to read • [Edit Online](#)

warning C6014: Leaking memory.

This warning indicates that the specified pointer points to allocated memory or some other allocated resource that has not been freed. The analyzer checks for this condition only when the `_Analysis_mode_(_Analysis_local_leak_checks_)` SAL annotation is specified. By default, this annotation is specified for Windows kernel mode (driver) code. For more information about SAL annotations, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

## Example

The following code generates this warning:

```
// cl.exe /analyze /EHsc /nologo /W4
#include <sal.h>
#include <stdlib.h>
#include <string.h>

_Analysis_mode_(_Analysis_local_leak_checks_)

#define ARRSIZE 10
const int TEST_DATA [ARRSIZE] = {10,20,30,40,50,60,70,80,90,100};

void f( )
{
    int *p = (int *)malloc(sizeof(int)*ARRSIZE);
    if (p) {
        memcpy(p, TEST_DATA, sizeof(int)*ARRSIZE);
        // code ...
    }
}

int main( )
{
    f();
}
```

## Example

The following code corrects the warning by releasing the memory:

```

// cl.exe /analyze /EHsc /nologo /W4
#include <sal.h>
#include <stdlib.h>
#include <string.h>

_Analysis_mode_(Analysis_local_leak_checks_)

#define ARRAYSIZE 10
const int TEST_DATA [ARRAYSIZE] = {10,20,30,40,50,60,70,80,90,100};

void f( )
{
    int *p = (int *)malloc(sizeof(int)*ARRAYSIZE);
    if (p) {
        memcpy(p, TEST_DATA, sizeof(int)*ARRAYSIZE);
        // code ...
        free(p);
    }
}

int main( )
{
    f();
}

```

This warning is reported for both memory and resource leaks when the resource is commonly *aliased* to another location. Memory is aliased when a pointer to the memory escapes the function by means of an `_out_` parameter annotation, global variable, or return value. This warning can be reported on function exit if the argument is annotated as having been expected to be released.

Note that Code Analysis will not recognize the actual implementation of a memory allocator (involving address arithmetic) and will not recognize that memory is allocated (although many wrappers will be recognized). In this case, the analyzer does not recognize that the memory was allocated and issues this warning. To suppress the false positive, use a `#pragma` directive on the line that precedes the opening brace `{` of the function body.

To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

```
// cl.exe /analyze /EHsc /nologo /W4
#include <sal.h>
#include <memory>

using namespace std;

_Analysis_mode_(_Analysis_local_leak_checks_)

const int ARRSIZE = 10;
const int TEST_DATA [ARRSIZE] = {10,20,30,40,50,60,70,80,90,100};

void f( )
{
    unique_ptr<int[]> p(new int[ARRSIZE]);
    std::copy(begin(TEST_DATA), end(TEST_DATA), p.get());

    // code ...

    // No need for free/delete; unique_ptr
    // cleans up when out of scope.
}

int main( )
{
    f();
}
```

## See Also

[C6211](#)

# C6029

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6029: possible buffer overrun in call to <function>: use of unchecked value

This warning indicates that a function that takes a buffer and a size is being passed a unchecked size. The data read-in from some external source has not been verified to see whether it is smaller than the buffer size. An attacker might intentionally specify a much larger than expected value for the size, which will lead to a buffer overrun.

Generally, whenever you read data from an untrusted external source, make sure to verify it for validity. It is usually appropriate to verify the size to make sure it is in the expected range.

## Example

The following code generates this warning by calling the annotated function `ReadFile` two times. After the first call, the Post attribute property marks the second parameter value untrusted. Therefore, passing an untrusted value in the second call to `ReadFile` generates this warning as shown in the following code:

```
#include "windows.h"

bool f(HANDLE hFile)
{
    char buff[MAX_PATH];

    DWORD cbLen;
    DWORD cbRead;

    // Read the number of byte to read (cbLen).
    if (!ReadFile (hFile, &cbLen, sizeof (cbLen), &cbRead, NULL))
    {
        return false;
    }
    // Read the bytes
    if (!ReadFile (hFile, buff, cbLen, &cbRead, NULL)) // warning 6029
    {
        return false;
    }

    return true;
}
```

To correct this warning, check the buffer size as shown in the following code:

```
bool f(HANDLE hFile)
{
    char buff[MAX_PATH];

    DWORD cbLen;
    DWORD cbRead;

    // Read the number of byte to read (cbLen).
    if (!ReadFile (hFile, &cbLen, sizeof (cbLen), &cbRead, NULL))
    {
        return false;
    }
    // Ensure that there's enough space in the buffer to read that many bytes.
    if (cbLen > sizeof(buff))
    {
        return false;
    }
    // Read the bytes
    if (!ReadFile (hFile, buff, cbLen, &cbRead, NULL)) // warning 6029
    {
        return false;
    }

    return true;
}
```

## See also

- [Using SAL Annotations to reduce code defects](#)

# C6031

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6031: return value ignored: <function> could return unexpected value

This warning indicates that the calling function is not checking the return value of a function call that signals failure via its return value. Depending on which function is being called, this defect can lead to seemingly random program misbehavior, including crashes and data corruptions in error conditions or low-resource situations.

In general, it is not safe to assume that a call to function requiring disk, network, memory, or other resources will always succeed. The caller should always check the return value and handle error cases appropriately. Also consider using the `_Must_inspect_result_` annotation, which checks that the value is examined in a useful way.

## Example

The following code generates this warning:

```
#include <stdio.h>
void f( )
{
    fopen( "test.c", "r" ); // return value ignored
    // code ...
}
```

To correct this warning, check the return value of the function as shown in the following code:

```
#include <stdio.h>
void f( )
{
    FILE *stream;
    if((stream = fopen( "test.c", "r" )) == NULL )
        return;
    // code ...
}
```

The following code uses safe function `fopen_s` to correct this warning:

```
#include <stdio.h>
void f( )
{
    FILE *stream;
    errno_t err;

    if( (err = fopen_s( &stream, "test.c", "r" )) !=0 )
    {
        // code ...
    }
}
```

This warning is also generated if the caller ignores the return value of a function annotated with the `_Check_return_` property as shown in the following code.

```
#include <sal.h>
_Check_return_ bool func();

void test_f()
{
    func(); // Warning C6031
}
```

To correct the previous warning, check the return value as shown in the following code:

```
#include <sal.h>
_Check_return_ bool func();

void test_f()
{
    if( func() ) {
        // code ...
    }
}
```

## See Also

- [fopen\\_s, \\_wfopen\\_s](#)
- [Using SAL Annotations to reduce code defects](#)

# C6053

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6053: call to <function> may not zero-terminate string <variable>

This warning indicates that the specified function has been called in such a way that the resulting string might not be zero-terminated. This defect might cause an exploitable buffer overrun or crash. This warning is also generated if an annotated function expects a null terminated string is passed a string that is not null terminated.

Most C standard library and Win32 string handling functions require and produce zero-terminated strings. A few 'counted string' functions (including `strncpy`, `wcsncpy`, `_mbsncpy`, `_snprintf`, and `snwprintf`) do not produce zero-terminated strings if they exactly fill their buffer. In this case, a subsequent call to a string function that expects a zero-terminated string will go beyond the end of the buffer looking for the zero. The program should make sure that the string ends with a zero. In general, you should pass a length to the 'counted string' function one smaller than the size of the buffer and then explicitly assign zero to the last character in the buffer.

## Example

The following sample code generates this warning:

```
#include <string.h>
#define MAX 15

size_t f( )
{
    char szDest[MAX];
    char *szSource="Hello, World!";

    strncpy(szDest, szSource, MAX);
    return strlen(szDest); // possible crash here
}
```

## Example

To correct this warning, zero-terminate the string as shown in the following sample code:

```
#include <string.h>
#define MAX 15

size_t f( )
{
    char szDest[MAX];
    char *szSource="Hello, World!";

    strncpy(szDest, szSource, MAX-1);
    szDest[MAX-1]=0;
    return strlen(szDest);
}
```

## Example

The following sample code corrects this warning using safe string manipulation `strncpy_s` function:

```
#include <string.h>
#define MAX 15

size_t f( )
{
    char szDest[MAX];
    char *szSource= "Hello, World!";

    strncpy_s(szDest, sizeof(szDest), szSource, strlen(szSource));
    return strlen(szDest);
}
```

You should note that this warning is sometimes reported on certain idioms guaranteed to be safe in practice. Because of the frequency and potential consequences of this defect, the analysis tool is biased in favor of finding potential issues instead of its typical bias of reducing noise.

## See Also

- [Using SAL Annotations to reduce code defects](#)
- [`strncpy\_s`, `\_strncpy\_s\_l`, `wcsncpy\_s`, `\_wcsncpy\_s\_l`, `\_mbsncpy\_s`, `\_mbsncpy\_s\_l`](#)

# C6054

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6054: string <variable> may not be zero-terminated

This warning indicates that a function that requires zero-terminated string was passed a non-zero terminated string. A function that expects a zero-terminated string will go beyond the end of the buffer to look for the zero. This defect might cause an exploitable buffer overrun error or crash. The program should make sure that the string ends with a zero.

## Example

The following code generates this warning:

```
#include <sal.h>

void func( _In_z_ wchar_t* wszStr );

void g( )
{
    wchar_t wcArray[200];
    func(wcArray); // Warning C6054
}
```

To correct this warning, null-terminate `wcArray` before calling function `func()` as shown in the following sample code:

```
#include <sal.h>

void func( _In_z_ wchar_t* wszStr );

void g( )
{
    wchar_t wcArray[200];
    wcArray[0]= '\0';
    func(wcArray);
}
```

## See Also

- [C6053](#)
- [Using SAL Annotations to reduce code defects](#)

# C6059

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6059: Incorrect length parameter in call to <function>. Pass the number of remaining characters, not the buffer size of <variable>

This warning indicates that a call to a string concatenation function is probably passing an incorrect value for the number of characters to concatenate. This defect might cause an exploitable buffer overrun or crash. A common cause of this defect is passing the buffer size, instead of the remaining number of characters in the buffer, to the string manipulation function.

## Example

The following code generates this warning:

```
#include <string.h>
#define MAX 25

void f( )
{
    char szTarget[MAX];
    char *szState = "Washington";
    char *szCity="Redmond, ";

    strncpy(szTarget,szCity, MAX);
    szTarget[MAX -1] = '\0';
    strncat(szTarget, szState, MAX); //wrong size
    // code ...
}
```

To correct this warning, use the correct number of characters to concatenate as shown in the following code:

```
#include <string.h>
#define MAX 25

void f( )
{
    char szTarget[MAX];
    char *szState = "Washington";
    char *szCity="Redmond, ";

    strncpy(szTarget,szCity, MAX);
    szTarget[MAX -1] = '\0';
    strncat(szTarget, szState, MAX - strlen(szTarget)); // correct size
    // code ...
}
```

To correct this warning using the safe string manipulation function, see the following code:

```
#include <string.h>

void f( )
{
    char *szState ="Washington";
    char *szCity="Redmond, ";

    size_t nTargetSize = strlen(szState) + strlen(szCity) + 1;
    char *szTarget= new char[nTargetSize];

    strncpy_s(szTarget, nTargetSize, szCity,strlen(szCity));
    strncat_s(szTarget, nTargetSize, szState,
              nTargetSize - strlen(szTarget));
    // code ...
    delete [] szTarget;
}
```

## See Also

- [\\_strncpy\\_s, \\_strncpy\\_s\\_l, \\_wcsncpy\\_s, \\_wcsncpy\\_s\\_l, \\_mbsncpy\\_s, \\_mbsncpy\\_s\\_l](#)
- [\\_strncat\\_s, \\_strncat\\_s\\_l, \\_wcsncat\\_s, \\_wcsncat\\_s\\_l, \\_mbsncat\\_s, \\_mbsncat\\_s\\_l](#)

# C6063

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6063: missing string argument to <function> corresponding to conversion specifier <number>

This warning indicates that not enough arguments are being provided to match a format string; at least one of the missing arguments is a string. This defect can cause crashes and buffer overflows (if the called function is of the `sprintf` family), as well as potentially incorrect output.

## Example

The following code generates this warning:

```
#include <string.h>
void f( )
{
    char buff[15];
    sprintf(buff, "%s %s", "Hello, World!");
}
```

To correct this warning, provide additional arguments as shown in the following code:

```
#include <string.h>
void f( )
{
    char buff[15];
    sprintf(buff, "%s %s ", "Hello","World");
}
```

The following code corrects this warning using safe string manipulation function:

```
#include <string.h>
void f( )
{
    char buff[15];
    sprintf_s( buff, sizeof(buff),"%"s", "Hello, World!" );
}
```

## See Also

[sprintf\\_s, \\_sprintf\\_s\\_l, swprintf\\_s, \\_swprintf\\_s\\_l](#)

# C6064

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6064: missing integer argument to <function> corresponding to conversion specifier <number>

This warning indicates that not enough arguments are being provided to match a format string and one of the missing arguments is an integer. This defect can cause incorrect output.

## Example

The following code generates this warning because an incorrect number of arguments were used in call to `sprintf` and the missing argument was an integer:

```
#include <string.h>
void f( )
{
    char buff[15];
    char *string="Hello, World";

    sprintf(buff,"%s %d", string);
}
```

To correct this warning, specify missing arguments as shown in the following code:

```
#include <string.h>
void f( )
{
    char buff[15];
    char *string = "Hello, World";

    sprintf(buff,"%s %d",string, strlen(string));
}
```

The following code uses safe string manipulation function, `sprintf_s` to correct this warning:

```
#include <string.h>
void f( )
{
    char buff[15];
    char *string="Hello World";

    sprintf_s(buff,sizeof(buff),"%s %d", string, strlen(string));
}
```

## See Also

[sprintf\\_s, \\_sprintf\\_s\\_l, swprintf\\_s, \\_swprintf\\_s\\_l](#)

# C6066

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6066: non-pointer passed as parameter <number> when pointer is required in call to <function>

This warning indicates that the format string specifies that a pointer is required, for example, a `%n` or `%p` specification for `printf` or a `%d` for `scanf`, but a non-pointer is being passed. This defect is likely to cause a crash or corruption of some form.

## Example

The following code generates this warning:

```
#include <stdio.h>
#define MAX 30
void f( )
{
    char buff[MAX];
    sprintf( buff, "%s %p %d", "Hello, World!", 1, MAX ); //warning C6066
    // code ...
}

void g( int i )
{
    int result;
    result = scanf( "%d", i ); // warning C6066
    // code ...
}
```

To correct this warning, the following code passes correct parameters to the `sprintf` and `scanf` functions:

```
#include <stdio.h>
#define MAX 30

void f( )
{
    char buff[MAX];

    sprintf( buff, "%s %p %d", "Hello, World!", buff, MAX ); // pass buff
    // code ...
}
void g( int i )
{
    int result;
    // code ...
    result = scanf( "%d", &i ); // pass the address of i
    // code ...
}
```

The following code use safe string manipulation functions — `sprintf_s` and `scanf_s` — to correct this warning:

```
void f( )
{
    char buff[MAX];

    sprintf_s( buff, sizeof(buff), "%s %p %d", "Hello, World!", buff, MAX );
    // code ...
}

void g( int i )
{
    int result;
    // code ...
    result = scanf_s( "%d", &i );
    // code ...
}
```

This warning is typically reported because an integer has been used for a `%p` format instead of a pointer. Using an integer in this instance is not portable to 64-bit computers.

## See Also

- [sprintf\\_s, \\_sprintf\\_s\\_l, swprintf\\_s, \\_swprintf\\_s\\_l](#)
- [scanf\\_s, \\_scanf\\_s\\_l, wscanf\\_s, \\_wscanf\\_s\\_l](#)

# C6067

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6067: parameter <number> in call to <function> must be the address of the string

This warning indicates a mismatch between the format specifier and the function parameter. Even though the warning suggests using the address of the string, you must check the type of parameter a function expects before correcting the problem. For example, a `%s` specification for `printf` requires a string argument, but a `%s` specification in `scanf` requires an address of the string.

This defect is likely to cause a crash or corruption of some form.

## Example

The following code generates this warning because an integer is passed instead of a string:

```
#include <stdio.h>
void f_defective( )
{
    char *str = "Hello, World!";
    printf("String:\n %s",1); // warning
    // code ...
}
```

To correct the warning, pass a string as a parameter to `printf` as shown in the following code:

```
#include <stdio.h>
void f_corrected( )
{
    char *str = "Hello, World!";
    printf("String:\n %s",str);
    // code ...
}
```

The following code generates this warning because an incorrect level of indirection is specified when passing the parameter, buffer, to `scanf`:

```
#include <stdio.h>
void h_defective( )
{
    int retval;
    char* buffer = new char(20);
    if ( buffer )
    {
        retval = scanf("%s", &buffer); // warning C6067
        // code...
        delete buffer ;
    }
}
```

To correct above warnings, pass the correct parameter as shown in the following code:

```
#include <stdio.h>
void h_corrected( )
{
    int retval;
    char* buffer = new char(20);
    if ( buffer )
    {
        retval = scanf("%s", buffer);
        // code...
        delete buffer;
    }
}
```

The following code uses safe string manipulation functions to correct this warning:

```
#include <stdio.h>
void f_safe( )
{
    char buff[20];
    int retVal;

    sprintf_s( buff, 20, "%s %s", "Hello", "World!" );
    printf_s( "String:\n  %s  %s", "Hello", "World!" );
    retVal = scanf_s("%s", buff, 20);
}
```

## See Also

- [printf\\_s, \\_printf\\_s\\_l, swprintf\\_s, \\_swprintf\\_s\\_l](#)
- [printf, \\_printf\\_l, wprintf, \\_wprintf\\_l](#)
- [scanf\\_s, \\_scanf\\_s\\_l, wscanf\\_s, \\_wscanf\\_s\\_l](#)

# C6101

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6101: Returning uninitialized memory

A successful path through the function does not set the named `_out_` parameter. This message is generated based on SAL annotations that indicate that the function in question always succeeds. A function that doesn't return a success/failure indication should set all of its `_out_` parameters because the analyzer assumes that the `_out_` parameter is uninitialized data before the function is called, and that the function will set the parameter so that it's no longer uninitialized. If the function does indicate success/failure, then the `_out_` parameter doesn't have to be set in the case of failure, and you can detect and avoid the uninitialized location. In either case, the objective is to avoid the reading of an uninitialized location. If the function sometimes doesn't touch an `_out_` parameter that's subsequently used, then the parameter should be initialized before the function call and be marked with the `_Inout_` annotation, or the more explicit `_Pre_null_` or `_Pre_satisfies_()` when appropriate. "Partial success" can be handled with the `_When_` annotation. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

# C6102

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6102: Using <variable> from failed function call at line <location>.

This warning is reported instead of [C6001](#) when a variable is not set because it was marked as an `_out_` parameter on a prior function call that failed.

The problem might be that the prior called function is not completely annotated. It may require `_Always_`, `_Outptr_result_nullonfailure_` (`_COM_Outptr_` for COM code), or a related annotation.

## See Also

- [C6001](#)
- [Using SAL Annotations to Reduce C/C++ Code Defects](#)

# C6103

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6103: Returning <variable> from failed function call at line <location>.

A successful path through the function is returning a variable that was used as an `_out_` parameter to an internal function call that failed.

The problem might be that the called function and the calling function are not completely annotated. The called function may require `_Always_`, `_Outptr_result_nullonfailure_` (`_COM_Outptr_` for COM code), or a related annotation, and the calling function may require a `_Success_` annotation. Another possible cause for this warning is that the `_Success_` annotation on the called function is incorrect.

## See Also

[Using SAL Annotations to Reduce C/C++ Code Defects](#)

# C6200

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6200: index <name> is out of valid index range <min> to <max> for non-stack buffer <variable>

This warning indicates that an integer offset into the specified array exceeds the maximum bounds of that array. This defect might cause random behavior or crashes.

One common cause of this defect is using the size of an array as an index into the array. Because C/C++ array indexing is zero-based, the maximum legal index into an array is one less than the number of array elements.

## Example

The following code generates this warning because the `for` loop exceeds the index range:

```
int buff[14]; // array of 0..13 elements
void f()
{
    for (int i=0; i<=14;i++) // i exceeds the index
    {
        buff[i]= 0; // warning C6200
        // code...
    }
}
```

To correct both warnings, use correct array size as shown in the following code:

```
int buff[14]; // array of 0..13 elements
void f()
{
    for ( int i=0; i < 14; i++) // loop stops when i < 14
    {
        buff[i]= 0; // initialize buffer
        // code...
    }
}
```

# C6201

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6201: buffer overrun for <variable>, which is possibly stack allocated: index <name> is out of valid index range <min> to <max>

This warning indicates that an integer offset into the specified stack array exceeds the maximum bounds of that array. This defect might cause random behavior or crashes.

One common cause of this defect is using an array's size as an index into the array. Because C/C++ array indexing is zero-based, the maximum legal index into an array is one less than the number of array elements.

## Example

The following code generates this warning because the array index is out of the valid range:

```
void f( )
{
    int buff[25];
    for (int i=0; i <= 25; i++) // i exceeds array bound
    {
        buff[i]=0; // initialize i
        // code ...
    }
}
```

To correct both warnings, use the correct array size as shown in the following code:

```
void f( )
{
    int buff[25];
    for (int i=0; i < 25; i++)
    {
        buff[i]=0; // initialize i
        // code ...
    }
}
```

# C6211

2/21/2019 • 2 minutes to read • [Edit Online](#)

warning C6211: Leaking memory <pointer> due to an exception. Consider using a local catch block to clean up memory

This warning indicates that allocated memory is not being freed when an exception is thrown. The statement at the end of the path could throw an exception. The analyzer checks for this condition only when the

`_Analysis_mode_(_Analysis_local_leak_checks_)` SAL annotation is specified. By default, this annotation is specified for Windows kernel mode (driver) code. For more information about SAL annotations, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

## Example

The following code generates this warning because an exception could be thrown during the second allocation and thereby leak the first allocation, or an exception could be thrown somewhere in the code that's represented by the "`code ...`" comment and thereby leak both allocations.

```
// cl.exe /analyze /c /EHsc /nologo /W4
#include <sal.h>

_Analysis_mode_(_Analysis_local_leak_checks_)
void f( )
{
    char *p1 = new char[10];
    char *p2 = new char[10];

    // code ...

    delete[] p2;
    delete[] p1;
}
```

To use the same allocation functions and correct this problem, add an exception handler:

```
// cl.exe /analyze /c /EHsc /nologo /W4
#include <sal.h>
#include <new>
#include <iostream>
using namespace std;

_Analysis_mode_(_Analysis_local_leak_checks_)

void f()
{
    char *p1 = nullptr;
    char *p2 = nullptr;

    try
    {
        p1 = new char[10];
        p2 = new char[10];

        // code ...

        delete [] p2;
        delete [] p1;
    }
    catch (const bad_alloc& ba)
    {
        cout << ba.what() << endl;
        delete [] p2;
        delete [] p1;
    }
    // code ...
}
```

To avoid these kinds of potential leaks altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

```
// cl.exe /analyze /c /EHsc /nologo /W4
#include <sal.h>
#include <vector>
#include <memory>

using namespace std;

_Analysis_mode_(_Analysis_local_leak_checks_)

void f( )
{
    // use 10-element vectors in place of char[10]
    vector<char> v1;
    vector<char> v2;

    for (int i=0; i<10; ++i) {
        v1.push_back('a');
        v2.push_back('b');
    }
    // code ...

    // use unique_ptr if you still want char[10]
    unique_ptr<char[]> a1(new char[10]);
    unique_ptr<char[]> a2(new char[10]);

    // code ...

    // No need for delete; vector and unique_ptr
    // clean up when out of scope.
}
```

## See Also

[C++ Exception Handling](#)

# C6214

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6214: cast between semantically different integer types: HRESULT to a Boolean type

This warning indicates that an `HRESULT` is being cast to a Boolean type. The success value (`s_OK`) of an `HRESULT` equals 0. However, 0 indicates failure for a Boolean type. Casting an `HRESULT` to a Boolean type and then using it in a test expression will yield an incorrect result. Sometimes, this warning occurs if an `HRESULT` is being stored in a Boolean variable. Any comparison that uses the Boolean variable to test for `HRESULT` success or failure could lead to incorrect results.

## Example

The following code generates this warning:

```
#include <windows.h>

BOOL f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;
    hr = CoGetMalloc(1, &pMalloc);
    if ((BOOL)hr) // warning 6214
    {
        // success code ...
        return TRUE;
    }
    else
    {
        // failure code ...
        return FALSE;
    }
}
```

To correct this warning, use the following code:

```
#include <windows.h>

BOOL f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (SUCCEEDED(hr))
    {
        // success code ...
        return TRUE;
    }
    else
    {
        // failure code ...
        return FALSE;
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

Usually, the `SUCCEEDED` or `FAILED` macro should be used to test the value of an `HRESULT`.

For more information, see one of the following topics:

[SUCCEEDED](#)

[FAILED](#)

To leverage modern C++ memory allocation methodology, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6215

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6215: cast between semantically different integer types: a Boolean type to HRESULT

This warning indicates that a Boolean is being cast to an `HRESULT`. Boolean types indicate success by a non-zero value, whereas success (`s_OK`) in `HRESULT` is indicated by a value of 0. Casting a Boolean type to an `HRESULT` and then using it in a test expression will yield an incorrect result.

This warning frequently occurs when a Boolean is used as an argument to `SUCCEEDED` or `FAILED` macro, which explicitly casts their arguments to an `HRESULT`.

## Example

The following code generates this warning:

```
#include <windows.h>
BOOL IsEqual(REFGUID, REFGUID);

void f( REFGUID riid1, REFGUID riid2 )
{
    if (SUCCEEDED( IsEqual( riid1, riid2 ) )) //warning 6215
    {
        // success code ...
    }
    else
    {
        // failure code ...
    }
}
```

Generally, the `SUCCEEDED` or `FAILED` macros should only be applied to `HRESULT`.

To correct this warning, use the following code:

```
#include <windows.h>
BOOL IsEqual(REFGUID, REFGUID);

void f( REFGUID riid1, REFGUID riid2 )
{
    if (IsEqual( riid1, riid2 ) == TRUE)
    {
        // code for riid1 == riid2
    }
    else
    {
        // code for riid1 != riid2
    }
}
```

For more information, see [SUCCEEDED Macro](#) and [FAILED Macro](#)

# C6216

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6216: compiler-inserted cast between semantically different integral types: a Boolean type to HRESULT

This warning indicates that a Boolean is being used as an `HRESULT` without being explicitly cast. Boolean types indicate success by a non-zero value; success (`S_OK`) in `HRESULT` is indicated by a value of 0. The typical failure value for functions that return a Boolean false is a success status when it is tested as an `HRESULT`. This is likely to lead to incorrect results.

## Example

The following code generates this warning:

```
#include <windows.h>
BOOL IsEqual(REFGUID, REFGUID);

HRESULT f( REFGUID riid1, REFGUID riid2 )
{
    // code ...
    return IsEqual(rIID1, rIID2);
}
```

To correct this warning, use the following code:

```
#include <windows.h>
BOOL IsEqual(REFGUID, REFGUID);

HRESULT f( REFGUID riid1, REFGUID riid2 )
{
    if (IsEqual(rIID1, rIID2) == TRUE)
    {
        // code ...
        return S_OK;
    }
    else
    {
        // code ...
        return E_FAIL;
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

For more information, see [SUCCEEDED Macro](#) and [FAILED Macro](#).

# C6217

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6217: Implicit cast between semantically different integer types: testing HRESULT with 'not'. Consider using `SUCCEEDED` or `FAILED` macro instead.

This warning indicates that an `HRESULT` is being tested with the not (`!`) operator. A success (`S_OK`) in `HRESULT` is indicated by a value of 0. However, 0 indicates failure for a Boolean type. Testing `HRESULT` with the not operator (`!`) to determine which code block to run can cause following the wrong code path. This will lead to unwanted results.

## Example

The following code generates this warning because the not operator is used to determine success or failure of an `HRESULT` value. In this case, wrong code path is executed because `( !hr )` runs the failure code:

```
#include <windows.h>
#include <objbase.h>

void f( )
{
    HRESULT hr = CoInitialize(NULL);
    if (!hr)
    {
        // failure code ...
    }
    else
    {
        // success code ...
    }
}
```

To correct this warning, the following code uses `FAILED` macro to look for failure:

```
#include <windows.h>
#include <objbase.h>

void f( )
{
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr))
    {
        // failure code ...
    }
    else
    {
        // success code ...
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

The typical success value of `HRESULT` (`S_OK`) is `false` when it is tested as a Boolean.

To verify whether `HRESULT` is a success, use the `SUCCEEDED` macro instead.

# C6219

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6219: Implicit cast between semantically different integer types: comparing HRESULT to 1 or TRUE.  
Consider using [SUCCEEDED](#) or [FAILED](#) macro instead

This warning indicates an `HRESULT` is being compared with an explicit, non-`HRESULT` value of one (1). This comparison is likely to lead to incorrect results, because the typical success value of `HRESULT` (`S_OK`) is 0. If you compare this value to a Boolean type it's implicitly converted to false.

## Example

The following code generates this warning because the `CoGetMalloc` returns an `HRESULT`, which then is compared to `TRUE`:

```
#include <windows.h>

void f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;
    hr = CoGetMalloc(1, &pMalloc);

    if (hr == TRUE)
    {
        // success code ...
    }
    else
    {
        // failure code
    }
}
```

Most of the time, this warning is caused by code that compares an `HRESULT` to a Boolean. It's better to use the [SUCCEEDED](#) or [FAILED](#) macros to test the value of an `HRESULT`. To correct this warning, use the following code:

```
#include <windows.h>

void f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;
    hr = CoGetMalloc(1, &pMalloc);

    if (SUCCEEDED(hr))
    {
        // success code ...
    }
    else
    {
        // failure code
    }
}
```

For this warning, the `SCODE` type is treated as an `HRESULT`.

The use of `malloc` and `free` (and related dynamic memory APIs) has many pitfalls as a cause of memory leaks and exceptions. To avoid these kinds of leaks and exception problems, use the pointer and container classes provided by the C++ Standard Library. These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6220

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6220 - Implicit cast between semantically different integer types: comparing HRESULT to -1. Consider using SUCCEEDED or FAILED macro instead

This warning indicates that an `HRESULT` is being compared with an explicit, non-`HRESULT` value of -1, which is not a well-formed `HRESULT`. A failure in `HRESULT` (`E_FAIL`) is not represented by a -1. Therefore, an implicit cast of an `HRESULT` to an integer will generate an incorrect value and is likely to lead to the wrong result.

## Example

In most cases, this warning is caused by the code mistakenly expecting that a function that should return an `HRESULT` instead returns an integer, by using -1 as a failure value. The following code sample generates this warning:

```
#include <windows.h>

HRESULT f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (hr == -1)
    {
        // failure code ...
        return E_FAIL;
    }
    else
    {
        // success code ...
        return S_OK;
    }
}
```

It is best to use the `SUCCEEDED` or `FAILED` macro to test the value of an `HRESULT`. To correct this warning, use the following code:

```
#include <windows.h>

HRESULT f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (FAILED(hr))
    {
        // failure code ...
        return E_FAIL;
    }
    else
    {
        // success code ...
        return S_OK;
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

Explicit comparison is appropriate to check for specific `HRESULT` values, such as, `E_FAIL`. Otherwise, use the `SUCCEEDED` or `FAILED` macros.

For more information, see [SUCCEEDED Macro](#) and [FAILED Macro](#).

Note that the use of malloc and free (and related dynamic memory allocation APIs) have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6221

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6221: Implicit cast between semantically different integer types: comparing HRESULT to an integer.  
Consider using SUCCEEDED or FAILED macros instead

This warning indicates that an `HRESULT` is being compared to an integer other than zero. A success in `HRESULT (S_OK)` is represented by a 0. Therefore, an implicit cast of an `HRESULT` to an integer will generate an incorrect value and is likely to lead to the wrong result. It is often caused by mistakenly expecting a function to return an integer when it actually returns an `HRESULT`.

## Example

The following code generates this warning by comparing `HRESULT` against an integer value:

```
#include <windows.h>

HRESULT f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (hr == 4)
    {
        // failure code ...
        return S_FALSE;
    }
    else
    {
        // success code ...
        return S_OK;
    }
}
```

To correct this warning, the following code uses the `FAILED` macro:

```
#include <windows.h>

HRESULT f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    hr = CoGetMalloc(1, &pMalloc);
    if (FAILED(hr))
    {
        // failure code ...
        return S_FALSE;
    }
    else
    {
        // success code ...
        return S_OK;
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

For more information, see [SUCCEEDED Macro](#) and [FAILED Macro](#).

Note that the use of malloc and free (and related dynamic memory allocation APIs) have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6225

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6225: Implicit cast between semantically different integer types: assigning 1 or TRUE to HRESULT.  
Consider using S\_FALSE instead

This warning indicates that an `HRESULT` is being assigned or initialized with a value of an explicit 1. Boolean types indicate success by a non-zero value; success (`S_OK`) in `HRESULT` is indicated by a value of 0. This warning is frequently caused by accidental confusion of Boolean and `HRESULT` types. To indicate success, the symbolic constant `S_OK` should be used.

## Example

In the following code, assignment of `HRESULT` generates this warning:

```
#include <windows.h>

VOID f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    if (SUCCEEDED(CoGetMalloc(1, &pMalloc)))
    {
        // code ...
        hr = S_OK;
    }
    else
    {
        // code ...
        hr = 1;
    }
}
```

To correct this warning, use the following code:

```
VOID f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    if (SUCCEEDED(CoGetMalloc(1, &pMalloc)))
    {
        hr = S_OK;
        // code ...
    }
    else
    {
        hr = S_FALSE;
        // code ...
    }
}
```

For this warning, the `SCODE` type is equivalent to `HRESULT`.

To indicate failure, `E_FAIL`, or another constant, should be used.

For more information see one of the following topics:

[SUCCEEDED](#)

[FAILED](#)

To leverage modern C++ memory allocation methodology, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6226

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6226: Implicit cast between semantically different integer types: assigning -1 to HRESULT. Consider using E\_FAIL instead.

This warning indicates that an `HRESULT` is assigned or initialized to an explicit value of -1. This warning is frequently caused by accidental confusion of integer and `HRESULT` types. To indicate success, use the symbolic constant `S_OK` instead. To indicate failure, use the symbolic constants that start with `E_constant`, such as `E_FAIL`.

For more information, see the [SUCCEEDED](#) and [FAILED](#) macros.

## Example

The following code generates this warning:

```
#include <windows.h>

VOID f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    if (FAILED(CoGetMalloc(1, &pMalloc)))
    {
        hr = -1;
        // code ...
    }
    else
    {
        // code ...
    }
}
```

To correct this warning, use the following code:

```
#include <windows.h>

VOID f( )
{
    HRESULT hr;
    LPMALLOC pMalloc;

    if (FAILED(CoGetMalloc(1, &pMalloc)))
    {
        hr = E_FAIL;
        // code ...
    }
    else
    {
        // code ...
    }
}
```

For this warning, the `SCODE` type is treated as an `HRESULT`.

The use of `malloc` and `free` (and related dynamic memory APIs) has many pitfalls as a cause of memory leaks

and exceptions. To avoid these kinds of leaks and exception problems, use the pointer and container classes provided by the C++ Standard Library. These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6230

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6230: implicit cast between semantically different integer types: using HRESULT in a Boolean context

This warning indicates that a bare `HRESULT` is used in a context where a Boolean result is expected, such as an `if` statement. This test is likely to yield incorrect results. For example, the typical success value for `HRESULT` (`S_OK`) is false when it's tested as a Boolean.

## Example

The following code generates this warning:

```
#include <windows.h>

VOID f( )
{
    LPMALLOC pMalloc;
    HRESULT hr = CoGetMalloc(1, &pMalloc);

    if (hr)
    {

        // code ...
    }
    else
    {
        // code ...
    }
}
```

In most situations, the `SUCCEEDED` or `FAILED` macro should be used to test the value of the `HRESULT`. To correct this warning, use the following code:

```
#include <windows.h>

VOID f( )
{
    LPMALLOC pMalloc;
    HRESULT hr = CoGetMalloc(1, &pMalloc);

    if (SUCCEEDED(hr))
    {

        // code ...
    }
    else
    {
        // code ...
    }
}
```

For this warning, the `SCODE` type is treated as an `HRESULT`.

The use of `malloc` and `free` (and related dynamic memory APIs) has many pitfalls as a cause of memory leaks and exceptions. To avoid these kinds of leaks and exception problems, use the pointer and container classes

provided by the C++ Standard Library. These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6235

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6235: (<non-zero constant> || <expression>) is always a non-zero constant

This warning indicates that a non-zero constant value, other than one, was detected on the left side of a logical-or operation that occurs in a test context. The right side of the logical-or operation is not evaluated because the resulting expression always evaluates to true. This is referred to as "short-circuit evaluation."

A non-zero constant value, other than one, suggests that the bitwise-AND operator (`&`) may have been intended. This warning is not generated for the common idiom when the non-zero constant is 1, because of its use for selectively enabling code paths, but it is generated if the non-zero constant evaluates to 1, for example `1+0`.

## Example

The following code generates this warning because `INPUT_TYPE` is 2:

```
#define INPUT_TYPE 2
void f(int n)
{
    if(INPUT_TYPE || n) //warning 6235 issued
    {
        puts("Always gets here");
    }
    else
    {
        puts("Never gets here");
    }
}
```

The following code uses the bitwise-AND (`&`) operator to correct this warning:

```
#define INPUT_TYPE 2
void f(int n)
{
    if((INPUT_TYPE & n) == 2)
    {
        puts("bitwise-AND comparison true");
    }
    else
    {
        puts("bitwise-AND comparison false");
    }
}
```

## See Also

[C Logical Operators](#)

# C6236

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6236: (<expression> || <non-zero constant>) is always a non-zero constant

This warning indicates that a non-zero constant value, other than one, was detected on the right side of a logical OR operation that occurs in a test context. Logically, this implies that the test is redundant and can be removed safely. Alternatively, it suggests that the programmer may have intended to use a different operator, for example, the equality ( == ), bitwise-AND ( & ) or bitwise-XOR ( ^ ) operator, to test for a specific value or flag.

This warning is not generated for the common idiom when the non-zero constant is 1, because of its use for selectively enabling code paths at compile time. However, the warning is generated if the non-zero constant is formed by an expression that evaluates to 1, for example, 1 + 0.

## Example

This code shows how warning C6236 can appear. Because `INPUT_TYPE` is not 0, the expression `n || INPUT_TYPE` is always non-zero, and the `else` clause is never executed. However, `INPUT_TYPE` is a constant with a value other than one, which suggests that it is meant as a value for comparison:

```
#define INPUT_TYPE 2
#include <stdio.h>

void f( int n )
{
    if ( n || INPUT_TYPE ) // analysis warning C6236
    {
        puts( "Always gets here" );
    }
    else
    {
        puts( "Never enters here" );
    }
}
```

The following code instead uses a bitwise-AND ( & ) operator to test whether the `INPUT_TYPE` bit of the input parameter `n` is set:

```
#define INPUT_TYPE 2
#include <stdio.h>

void f( int n )
{
    if ( n & INPUT_TYPE ) // no warning
    {
        puts( "Bitwise-AND comparison is true" );
    }
    else
    {
        puts( "Bitwise-AND comparison is false" );
    }
}
```

## See Also



# C6237

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6237: (<zero> && <expression>) is always zero. <expression> is never evaluated and may have side effects

This warning indicates that a constant value of zero was detected on the left side of a logical-and operation that occurs in a test context. The resulting expression always evaluates to false. Therefore, the right side of the logical-AND operation is not evaluated. This is referred to as "short-circuit evaluation."

You should examine the right side of the expression carefully to ensure that any side effects such as assignment, function call, increment, and decrement operations needed for proper functionality are not affected by the short-circuit evaluation.

The expression (`0 && n`) produces no side effects and is commonly used to selectively choose code paths.

## Example

The following code shows various code samples that generate this warning:

```

#include <stdio.h>
#define INPUT_TYPE 0

int test();

// side effect: n not incremented
void f1( int n )
{
    if(INPUT_TYPE && n++) //warning: 6237
    {
        puts("code path disabled");
    }
    else
    {
        printf_s("%d - n was not incremented",n);
    }
}

// side effect: test() not called
void f2( )
{
    if(INPUT_TYPE && test()) //warning: 6237
    {
        puts("code path disabled");
    }
    else
    {
        puts("test() was not called");
    }
}

//side effect: assignment and function call did not occur
void f3( int n )
{
    if(INPUT_TYPE && ( n=test() ) ) //warning: 6237
    {
        puts("code path disabled");
    }
    else
    {
        printf_s("%d -- n unchanged. test() was not called", n);
    }
}

```

To correct this warning, use the following code:

```
#include <stdio.h>
#define INPUT_TYPE 0
int test();

void f1( int n )
{
if(INPUT_TYPE)
{
    if(n++)
    {
        puts("code path disabled");
    }
}
else
{
    puts("n was not incremented");
}
}

void f2( )
{
if(INPUT_TYPE)
{
    if( test() )
    {
        puts("code path disabled");
    }
}
else
{
    puts("test() was not called");
}
}

void f3( int n )
{
if(INPUT_TYPE)
{
    n = test();
    if( n )
    {
        puts("code path disabled");
    }
}
else
{
    puts("test() was not called");
}
}
```

## See Also

[C Logical Operators](#)

# C6239

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6239: (<non-zero constant> && <expression>) always evaluates to the result of <expression>. Did you intend to use the bitwise-and operator?

This warning indicates that a non-zero constant value, other than one, was detected on the left side of a logical-AND operation that occurs in a test context. For example, the expression `( 2 && n )` is reduced to `(!!n)`, which is the Boolean value of `n`.

This warning typically indicates an attempt to check a bit mask in which the bitwise-AND (`&`) operator should be used, and is not generated if the non-zero constant evaluates to 1 because of its use for selectively choosing code paths.

## Example

The following code generates this warning:

```
#include <stdio.h>
#define INPUT_TYPE 2
void f( int n )
{
    if(INPUT_TYPE && n) // warning 6239
    {
        puts("boolean value of n is true");
    }
    else
    {
        puts("boolean value of n is false");
    }
}
```

To correct this warning, use bitwise-AND (`&`) operator as shown in the following code:

```
#include <stdio.h>
#define INPUT_TYPE 2
void f( int n )
{
    if( ( INPUT_TYPE & n ) )
    {
        puts("bitmask true");
    }
    else
    {
        puts("bitmask false");
    }
}
```

## See Also

[& Operator](#)

# C6240

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6240: (<expression> && <non-zero constant>) always evaluates to the result of <expression>. Did you intend to use the bitwise-and operator?

This warning indicates that a non-zero constant value, other than one, was detected on the right side of a logical-and operation that occurs in a test context. For example, the expression `(n && 3)` reduces to `(!n)`, which is the Boolean value of `n`.

This warning typically indicates an attempt to check a bit mask in which the bitwise-AND (`&`) operator should be used. It is not generated if the non-zero constant evaluates to 1 because of its use for selectively choosing code paths.

## Example

The following code generates this warning:

```
#include <stdio.h>
#define INPUT_TYPE 2

void f(int n)
{
    if (n && INPUT_TYPE)
    {
        puts("boolean value of !n is true");
    }
    else
    {
        puts("boolean value of !n is false");
    }
}
```

To correct this warning, use bitwise-AND operator as shown in the following code:

```
#include <stdio.h>
#define INPUT_TYPE 2

void f(int n)
{
    if ( (n & INPUT_TYPE) )
    {
        puts("bitmask true");
    }
    else
    {
        puts("bitmask false");
    }
}
```

## See Also

[& Operator](#)

# C6242

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6242: A jump out of this try-block forces local unwind. Incurs severe performance penalty

This warning indicates that a jump statement causes control-flow to leave the protected block of a try-finally other than by fall-through.

Leaving the protected block of a try-finally other than by falling through from the last statement requires local unwind to occur. Local unwind typically requires approximately 1000 machine instructions; therefore, it is detrimental to performance.

Use `_leave` to exit the protected block of a try-finally.

## Example

The following code generates this warning:

```
#include <malloc.h>
void DoSomething(char *p); // function can throw exception

int f( )
{
    char *ptr = 0;
    __try
    {
        ptr = (char*) malloc(10);
        if ( !ptr )
        {
            return 0;    //Warning: 6242
        }
        DoSomething( ptr );
    }
    __finally
    {
        free( ptr );
    }
    return 1;
}
```

To correct this warning, use `_leave` as shown in the following code:

```
#include <malloc.h>
void DoSomething(char *p);
int f()
{
    char *ptr = 0;
    int retVal = 0;

    __try
    {
        ptr = (char *) malloc(10);
        if ( !ptr )
        {
            retVal = 0;
            __leave; //No warning
        }
        DoSomething( ptr );
        retVal = 1;
    }
    __finally
    {
        free( ptr );
    }

    return retVal;
}
```

The use of `malloc` and `free` have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

## See Also

[try-finally Statement](#)

# C6244

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6244: local declaration of <variable> hides previous declaration at <line> of <file>

This warning indicates that a declaration has the same name as a declaration at an outer scope and hides the previous declaration. You will not be able to refer to the previous declaration from inside the local scope. Any intended use of the previous declaration will end up using the local declaration. This warning only identifies a scope overlap and not lifetime overlap.

## Example

The following code generates this warning:

```
#include <stdlib.h>
#pragma warning(push)

// disable warning C4101: unreferenced local variable
#pragma warning(disable: 4101)

int i;
void f();
void (*pf)();

void test()
{
    // Hide global int with local function pointer
    void (*i)(); //Warning: 6244

    // Hide global function pointer with an int
    int pf; //Warning: 6244
}
#pragma warning(pop)
```

To correct this warning, use the following sample code:

```
#include <stdlib.h>
#pragma warning(push)
// disable warning C4101: unreferenced local variable
#pragma warning(disable: 4101)

int g_i;           // modified global variable name
void g_f();        // modified global function name
void (*f_pf)();   // modified global function pointer name

void test()
{
    void (*i)();
    int pf;
}
#pragma warning(pop)
```

When dealing with memory allocation, review code to determine whether an allocation was saved in one variable and freed by another variable.

# C6246

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6246: Local declaration of <variable> hides declaration of same name in outer scope. Additional Information: See previous declaration at <location>.

This warning indicates that two declarations have the same name at local scope. The name at outer scope is hidden by the declaration at the inner scope. Any intended use of the outer scope declaration will result in the use of local declaration.

## Example

The following code generates this warning:

```
#include <stdlib.h>
#define UPPER_LIMIT 256
int DoSomething( );

int f( )
{
    int i = DoSomething( );
    if (i > UPPER_LIMIT)
    {
        int i;
        i = rand( );
    }
    return i;
}
```

To correct this warning, use another variable name as shown in the following code:

```
#include <stdlib.h>
#define UPPER_LIMIT 256
int DoSomething( );

int f ( )
{
    int i = DoSomething( );
    if (i > UPPER_LIMIT)
    {
        int j = rand( );
        return j;
    }
    else
    {
        return i;
    }
}
```

This warning only identifies a scope overlap.

# C6248

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6248: setting a SECURITY\_DESCRIPTOR's DACL to NULL will result in an unprotected object

This warning identifies a call that sets a SECURITY\_DESCRIPTOR's DACL field to null. If the DACL that belongs to the security descriptor of an object is set to NULL, a null DACL is created. A null DACL grants full access to any user who requests it; normal security checking is not performed with respect to the object. A null DACL should not be confused with an empty DACL. An empty DACL is a properly allocated and initialized DACL that contains no ACEs. An empty DACL grants no access to the object it is assigned to.

Objects that have null DACLs can have their security descriptors altered by malicious users so that no one has access to the object.

Even if everyone needs access to an object, the object should be secured so that only administrators can alter its security. If only the creator needs access to an object, a DACL should not be set on the object; the system will choose an appropriate default.

## Example

The following code generates this warning because a null DACL is passed to the `SetSecurityDescriptorDacl` function:

```
#include <windows.h>

void f( PSECURITY_DESCRIPTOR pSecurityDescriptor )
{
    if (SetSecurityDescriptorDacl(pSecurityDescriptor,
                                TRUE,          // Dacl Present
                                NULL,          // NULL pointer to DACL
                                FALSE))        // Defaulted

    {
        // Dacl is now applied to an object
    }
}
```

To see a complete example on how to create security descriptor, see [Creating a Security Descriptor for a New Object in C++](#). For more information, see [Creating a DACL](#).

# C6250

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6250: Calling <function> VirtualFree without the MEM\_RELEASE flag may free memory but not address descriptors (VADs); results in address space leaks

This warning indicates that a call to `VirtualFree` without the `MEM_RELEASE` flag only decommits the pages, and does not release them. To decommit and release pages, use `MEM_RELEASE` flag in call to `VirtualFree`. If any pages in the region are committed, the function first decommits and then releases them. After this operation, the pages are in the free state. If you specify this flag, `dwSize` must be zero, and `lpAddress` must point to the base address returned by the `VirtualAlloc` function when the region was reserved. The function fails if either of these conditions is not met.

You can ignore this warning if your code later frees the address space by calling `VirtualFree` with the `MEM_RELEASE` flag.

For more information see [VirtualAlloc](#) and [VirtualFree](#).

The use of `VirtualAlloc` and `VirtualFree` have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

## Example

The following sample code generates this warning:

```

#include <windows.h>
#include <stdio.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( )
{
    LPVOID lpvBase; // base address of the test memory
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo(&sSysInfo);
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS);

    // code to access memory
    // ...

    if (lpvBase != NULL)
    {
        if (VirtualFree( lpvBase, 0, MEM_DECOMMIT )) // decommit pages
        {
            puts ("MEM_DECOMMIT Succeeded");
        }
        else
        {
            puts("MEM_DECOMMIT failed");
        }
    }
    else
    {
        puts("lpvBase == NULL");
    }
}

```

To correct this warning, use the following sample code:

```
#include <windows.h>
#include <stdio.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( )
{
    LPVOID lpvBase; // base address of the test memory
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo(&sSysInfo);
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS);

    //
    // code to access memory
    //

    if (lpvBase != NULL)
    {
        if (VirtualFree(lpvBase, 0, MEM_RELEASE)) // decommit & release
        {
            // code ...
        }
        else
        {
            puts("MEM_RELEASE failed");
        }
    }
    else
    {
        puts("lpvBase == Null ");
        // code...
    }
}
```

# C6255

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6255: `_alloca` indicates failure by raising a stack overflow exception. Consider using `_malloca` instead

This warning indicates that a call to `_alloca` has been detected outside of local exception handling. `_alloca` should always be called from within the protected range of an exception handler because it can raise a stack overflow exception on failure. If possible, instead of using `_alloca`, consider using `_malloca` which is a more secure version of `_alloca`.

## Example

The following code generates this warning because `_alloca` can generate exception:

```
#include <windows.h>

void f( )
{
    void *p = _alloca(10);
    // code ...
}
```

To correct this warning, use `_malloca` and add exception handler as shown in the following code:

```
#include <windows.h>
#include <malloc.h>

void f( )
{
    void *p;
    int errcode;
    __try
    {
        p = _malloca(10);
        // code...
        _freea(p);
    }
    __except( (GetExceptionCode() == STATUS_STACK_OVERFLOW ) ?
              EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH )
    {
        errcode = _resetstkoflw();
        // code ...
    }
}
```

## See Also

[\\_malloca](#)

# C6258

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning C6258: using TerminateThread does not allow proper thread clean up.

This warning indicates that a call to TerminateThread has been detected.

TerminateThread is a dangerous function that should only be used in the most extreme cases. For more information about problems associated with TerminateThread call, see [TerminateThread function](#).

## To properly terminate threads

1. Create an event object using the `CreateEvent` function.
2. Create the threads.
3. Each thread monitors the event state by calling the `WaitForSingleObject` function.
4. Each thread ends its own execution when the event is set to the signaled state (`WaitForSingleObject` returns `WAIT_OBJECT_0`).

## See also

- [Terminating a Thread](#)
- [WaitForSingleObject](#)
- [SetEvent](#)

# C6259

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6259: labeled code is unreachable: (<expression> & <constant>) in switch-expr cannot evaluate to <case-label>

This warning indicates unreachable code caused by the result of a bitwise-AND (`&`) comparison in a switch expression. The case statement that matches the constant in the switch expression is only reachable; all other case statements are not reachable.

## Example

The following sample code generates this warning because the expression `switch``(rand() & 3)` cannot evaluate to case label (`case 4`):

```
#include <stdlib.h>

void f()
{
    switch (rand () & 3) {
        case 3:
            /* Reachable */
            break;
        case 4:
            /* Not reachable */
            break;
        default:
            break;
    }
}
```

To correct this warning, remove the unreachable code or verify that the constant used in the case statement is correct. The following code removes the unreachable case statement:

```
#include <stdlib.h>

void f()
{
    switch (rand () & 3) {
        case 3:
            /* Reachable */
            break;
        default:
            break;
    }
}
```

## See Also

- [switch Statement \(C++\)](#)
- [switch Statement](#)

# C6260

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6260: sizeof \* sizeof is almost always wrong, did you intend to use a character count or a byte count?

This warning indicates that the results of two `sizeof` operations have been multiplied together. The C/C++ `sizeof` operator returns the number of bytes of storage an object uses. It is typically incorrect to multiply it by another `sizeof` operation; usually one is interested in the number of bytes in an object or the number of elements in an array (for example the number of wide-characters in an array).

There is some unintuitive behavior associated with `sizeof` operator. For example, in C, the `sizeof ('\0') == 4`, because a character is of an integral type. In C++, the type of a character literal is `char`, so `sizeof ('\0') == 1`. However, in both C and C++, the following is true:

```
sizeof ("\0") == 2.
```

## Example

The following code generates this warning:

```
#include <windows.h>

void f( )
{
    int i;
    i = sizeof(L"String") * sizeof(WCHAR);
    // code ...
}
```

To correct this warning, use the following code:

```
#include <windows.h>

void f( )
{
    int i;
    i= sizeof(L"String") / sizeof(WCHAR);

    /* or to get bytes */
    i = sizeof(L"String");
    // code ...
}
```

## See Also

- [sizeof Operator](#)
- [sizeof Operator \(C\)](#)

# C6262

2/21/2019 • 2 minutes to read • [Edit Online](#)

warning C6262: Function uses <constant> bytes of stack: exceeds /analyze:stacksize<constant 2>. Consider moving some data to heap

This warning indicates that stack usage that exceeds a preset threshold (`<constant 2>`) has been detected in a function. The default stack frame size for this warning is 16 KB for user mode, 1 KB for kernel mode. Stack—even in user mode—is limited, and failure to commit a page of stack causes a stack overflow exception. Kernel mode has a 12 KB stack size limit, which cannot be increased; therefore, kernel-mode code should aggressively limit stack use.

To correct the problem behind this warning, you can either move some data to the heap or to other dynamic memory. In user mode, one large stack frame may not be a problem—and this warning may be suppressed—but a large stack frame increases the risk of a stack overflow. (A large stack frame might occur if the function uses the stack heavily or is recursive.) The total stack size in user mode can be increased if stack overflow actually occurs, but only up to the system limit. You can use the **/analyze** command-line option to change the value for `<constant 2>`, but increasing it introduces a risk that an error will not be reported.

For kernel-mode code—for example, in driver projects—the value of `<constant 2>` is set to 1 KB. Well-written drivers should have very few functions that approach this value, and changing the limit downward may be desirable. The same general techniques that are used for user-mode code to reduce the stack size can be adapted to kernel-mode code.

## Example

The following code generates this warning because `char buffer` allocates 16,382 bytes, and the local integer variable `i` allocates another 4 bytes, which together exceed the default stack size limit of 16 KB.

```
// cl.exe /c /analyze /EHsc /W4
#include <windows.h>
#define MAX_SIZE 16382

void f( )
{
    int i;
    char buffer[MAX_SIZE];

    i = 0;
    buffer[0]='\0';

    // code...
}
```

The following code corrects this warning by moving some data to heap.

```
// cl.exe /c /analyze /EHsc /W4
#include <stdlib.h>
#include <malloc.h>
#define MAX_SIZE 16382

void f( )
{
    int i;
    char *buffer;

    i = 0;
    buffer = (char *) malloc( MAX_SIZE );
    if (buffer != NULL)
    {
        buffer[0] = '\0';
        // code...
        free(buffer);
    }
}
```

The use of malloc and free have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

### To correct this warning by adjusting the stack size

1. On the menu bar, choose **Project, Properties**.

The **Property Pages** dialog box is displayed.

2. Expand **Configuration Properties**.
3. Expand **C/C++**.
4. Select **Command Line** properties.
5. In **Additional options**, add **/analyze:stacksize16388**.

## See Also

- [/STACK \(Stack Allocations\)](#)
- [\\_resetstkoflw](#)
- [How to: Use Native Run-Time Checks](#)

# C6263

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6263: using `_alloca` in a loop; this can quickly overflow stack

This warning indicates that calling `_alloca` inside a loop to allocate memory can cause stack overflow. `_alloca` allocates memory from the stack, but that memory is only freed when the calling function exits. Stack, even in user-mode, is limited, and failure to commit a page of stack causes a stack overflow exception. The `_resetstkoflw` function recovers from a stack overflow condition, allowing a program to continue instead of failing with a fatal exception error. If the `_resetstkoflw` function is not called, there is no guard page after the previous exception. The next time that there is a stack overflow, there are no exceptions at all and the process terminates without warning.

You should avoid calling `_alloca` inside a loop if either the allocation size or the iteration count is unknown because it might cause stack overflow. In these cases, consider other options such as, heap memory, or [C++ Standard Library](#) classes.

## Example

The following code generates this warning:

```
#include <windows.h>
#include <malloc.h>
#include <excpt.h>
#include <stdio.h>

#define MAX_SIZE 50

void f ( int size )
{
    char* cArray;
    __try
    {
        for(int i = 0; i < MAX_SIZE; i++)
        {
            cArray = (char *)_alloca(size);

            // process cArray...
        }
    }
    __except(GetExceptionCode() == STATUS_STACK_OVERFLOW ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH )
    {
        // code...
        puts("Allocation Failed");
        _resetstkoflw();
    }
}
```

The following code uses `malloc()` to correct this warning:

```
#include <windows.h>
#define MAX_SIZE 50

void f ( int size )
{
    char* cArray;

    for(int i = 0; i < MAX_SIZE; i++)
    {
        cArray = (char *) malloc(size);
        if (cArray != NULL)
        {
            // process cArray...
            free(cArray);
        }
    }
}
```

## See Also

- [malloc](#)
- [\\_alloca](#)
- [\\_malloca](#)
- [C++ Standard Library](#)

# C6268

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6268: Incorrect order of operations: (<TYPE1>)(<TYPE2>)x + y. Possible missing parentheses in (<TYPE1>)((<TYPE2>)x + y)

This warning indicates that a complex cast expression might involve a precedence problem when performing pointer arithmetic. Because casts group more closely than binary operators, the result might not be what the programmer intended. In some cases, this defect causes incorrect behavior or a program crash.

In an expression such as:

```
(char *)p + offset
```

the offset is interpreted as an offset in characters; however, an expression such as:

```
(int *)(char *)p + offset
```

is equivalent to:

```
((int *)(char *)p) + offset
```

and so the offset is interpreted as an offset in integers. In other words, it is equivalent to:

```
(int *)((char *)p + (offset * sizeof(int)))
```

which is not likely to be what the programmer intended.

Depending on the relative sizes of the two types, this can lead to a buffer overrun.

## Example

The following code generates this warning:

```
void f(int *p, int offset_in_bytes)
{
    int *ptr;
    ptr = (int *)(char *)p + offset_in_bytes;
    // code ...
}
```

To correct this warning, use the following code:

```
void f(int *p, int offset_in_bytes)
{
    int *ptr;
    ptr = (int *)((char *)p + offset_in_bytes);
    // code ...
}
```

# C6269

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6269: possible incorrect order of operations: dereference ignored

This warning indicates that the result of a pointer dereference is being ignored, which raises the question of why the pointer is being dereferenced in the first place.

The compiler will correctly optimize away the gratuitous dereference. In some cases, however, this defect may reflect a precedence or logic error.

One common cause for this defect is an expression statement of the form:

```
*p++;
```

If the intent of this statement is simply to increment the pointer `p`, then dereference is unnecessary; however, if the intent is to increment the location that `p` is pointing to, then the program will not behave as intended because `p++` construct is interpreted as `(p++)` instead of `(*p)++`.

## Example

The following code generates this warning:

```
#include <windows.h>

void f( int *p )
{
    // code ...
    if( p != NULL )
        *p++;
    // code ...
}
```

To correct this warning, use parentheses as shown in the following code:

```
#include <windows.h>

void f( int *p )
{
    // code ...
    if( p != NULL )
        (*p)++;
    // code ...
}
```

# C6270

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6270: missing float argument to <function>: add a float argument corresponding to conversion specifier <number>

This warning indicates that not enough arguments are being provided to match a format string; at least one of the missing arguments is a floating-point number. This defect can lead to crashes, in addition to potentially incorrect output.

## Example

The following code generates this warning:

```
#include <stdio.h>
#include <string.h>

void f()
{
    char buff [25];
    sprintf(buff,"%s %f","pi:");
}
```

To correct this warning, pass the missing argument as shown in the following code:

```
#include <stdio.h>
#include <string.h>

void f()
{
    char buff [25];
    sprintf(buff,"%s %f","pi:",3.1415);
}
```

The following sample code uses the safe string manipulation function, `sprintf_s`, to correct this warning:

```
#include <stdio.h>
#include <string.h>

void f()
{
    char buff [25];
    sprintf_s( buff, 25,"%s %f", "pi:",3.1415 );
}
```

## See Also

[sprintf](#), [\\_sprintf\\_l](#), [swprintf](#), [\\_swprintf\\_l](#), [\\_\\_swprintf\\_l](#)

# C6271

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6271: extra argument passed to <function>: parameter <number> is not used by the format string

This warning indicates that additional arguments are being provided beyond those specified by the format string. By itself, this defect will not have any visible effect although it indicates that the programmer's intent is not reflected in the code.

## Example

The following sample code generates this warning:

```
#include <stdio.h>
#include <string.h>

void f()
{
    char buff[5];

    sprintf(buff,"%d",1,2);
}
```

To correct this warning, use the following sample code:

```
#include <stdio.h>
#include <string.h>

void f()
{
    char buff[5];

    sprintf(buff,"%d, %d",1,2);
}
```

The following sample code calls the safe string manipulation function, `sprintf_s`, to correct this warning:

```
#include <stdio.h>
#include <string.h>

void f()
{
    char buff[5];

    sprintf_s( buff, 5,"%s %d", 1,2 ); //safe version
}
```

## See Also

[sprintf](#), [\\_sprintf\\_l](#), [swprintf](#), [\\_swprintf\\_l](#), [\\_\\_swprintf\\_l](#)

# C6272

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6272: non-float passed as argument <number> when float is required in call to <function>

This warning indicates that the format string specifies that a float is required, for example, a `%f` or `%g` specification for `printf`, but a non-float such as an integer or string is being passed. This defect is likely to result in incorrect output; however, in certain circumstances it could result in a crash.

## Example

The following code generates this warning:

```
#include <stdio.h>
#include <string.h>

void f()
{
    char buff[5];
    int i=5;

    sprintf(buff,"%s %f","a",i);
}
```

To correct this warning, use `%i` instead of `%f` specification as shown in the following code:

```
#include <stdio.h>
#include <string.h>

void f()
{
    char buff[5];
    int i=5;

    sprintf(buff,"%s %i","a",i);
}
```

The following code uses the safe string manipulation function, `sprintf_s`, to correct this warning:

```
#include <stdio.h>
#include <string.h>

void f()
{
    char buff[5];
    int i=5;

    sprintf_s(buff,5,"%s %i","a",i); // safe version
}
```

## See Also

[sprintf](#), [\\_sprintf\\_l](#), [swprintf](#), [\\_swprintf\\_l](#), [\\_\\_swprintf\\_l](#)

# C6273

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning 6273 - non-integer passed as parameter <number> when integer is required in call to <function>; if a pointer value is being passed, %p should be used

This warning indicates that the format string specifies an integer, for example, a `%d`, length or precedence specification for `printf` but a non-integer such as a `float`, string, or `struct` is being passed as a parameter. This defect is likely to result in incorrect output.

## Example

The following code generates this warning because an integer is required instead of a `float` to `sprintf` function:

```
#include <stdio.h>
#include <string.h>

void f_defective()
{
    char buff[50];
    float f=1.5;

    sprintf(buff, "%d",f);
}
```

The following code uses an integer cast to correct this warning:

```
#include <stdio.h>
#include <string.h>

void f_corrected()
{
    char buff[50];
    float f=1.5;

    sprintf(buff,"%d",(int)f);
}
```

The following code uses safe string manipulation function, `sprintf_s`, to correct this warning:

```
#include <stdio.h>
#include <string.h>

void f_safe()
{
    char buff[50];
    float f=1.5;

    sprintf_s(buff,50,"%d",(int)f);
}
```

This warning is not applicable on Windows 9x and Windows NT version 4 because %p is not supported on these platforms.

## See Also

[sprintf](#), [\\_sprintf\\_l](#), [swprintf](#), [\\_swprintf\\_l](#), [\\_\\_swprintf\\_l](#)

# C6274

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6274: non-character passed as parameter <number> when character is required in call to <function>

This warning indicates that the format string specifies that a character is required (for example, a `%c` or `\%c` specification) but a non-integer such as a float, string, or struct is being passed. This defect is likely to cause incorrect output.

## Example

The following code generates this warning:

```
#include <stdio.h>
#include <string.h>

void f(char str[])
{
    char buff[5];

    sprintf(buff,"%c",str);
}
```

To correct this warning, use the following code:

```
#include <stdio.h>
#include <string.h>

void f(char str[])
{
    char buff[5];

    sprintf(buff,"%c",str[0]);
}
```

The following code uses safe string manipulation function, `sprintf_s`, to correct this warning:

```
#include <stdio.h>
#include <string.h>

void f(char str[])
{
    char buff[5];

    sprintf_s(buff,5,"%c", str[0]);
}
```

# C6276

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6276: Cast between semantically different string types: `char*` to `wchar_t*`. Use of invalid string can lead to undefined behavior

This warning indicates a potentially incorrect cast from an ANSI string (`char_t*`) to a UNICODE string (`wchar_t *`). Because UNICODE strings have a character size of 2 bytes, this cast might yield strings that are not correctly terminated. Using such strings with the `wcs*` library of functions could cause buffer overruns and access violations.

## Example

The following code generates this warning:

```
#include <windows.h>
VOID f()
{
    WCHAR szBuffer[8];
    LPWSTR pSrc;

    pSrc = (LPWSTR)"a";
    wcscpy(szBuffer, pSrc);
}
```

The following code corrects this warning by appending the letter L to represent the ASCII character as a wide character:

```
#include <windows.h>

VOID f()
{
    WCHAR szBuffer[8];
    LPWSTR pSrc;

    pSrc = L"a";
    wcscpy(szBuffer, pSrc);
}
```

The following code uses the safe string manipulation function, `wcscpy_s`, to correct this warning:

```
#include <windows.h>

VOID f()
{
    WCHAR szBuffer[8];
    LPWSTR pSrc;
    pSrc = L"a";
    wcscpy_s(szBuffer,8,pSrc);
}
```

# C6277

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6277: NULL application name with an unquoted path in call to <function>: results in a security vulnerability if the path contains spaces

This warning indicates that the application name parameter is null and there might be spaces in the executable path name. In this case, unless the executable name is "fully qualified," there is likely to be a security problem. A malicious user might insert a rogue executable with the same name earlier in the path. To correct this warning, you can specify the application name instead of passing null or if you do pass null for the application name, use quotation marks around the executable path.

## Example

The following sample code generates this warning because the application name parameter is null, and the executable path name has a space in it; there is a risk that a different executable could be run because of the way the function parses spaces. For more information, see [CreateProcess](#).

```
#include <windows.h>
#include <stdio.h>

void f_defective()
{
    STARTUPINFOA si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof( si ) );
    si.cb = sizeof( si );
    ZeroMemory( &pi, sizeof( pi ) );
    if( !CreateProcessA( NULL,
                        "C:\\Program Files\\\\MyApp",
                        NULL,
                        NULL,
                        FALSE,
                        0,
                        NULL,
                        NULL,
                        &si,
                        &pi ) )
    {
        puts( "CreateProcess failed." );
        return;
    }
    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

To correct this warning, use quotation marks around the executable path, as shown in the following example:

```
#include <windows.h>
#include <stdio.h>

void f ()
{
    STARTUPINFOA si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof( si ) );
    si.cb = sizeof( si );
    ZeroMemory( &pi, sizeof( pi ) );

    if( !CreateProcessA( NULL,
                        "\"C:\\Program Files\\MyApp.exe\"",
                        NULL,
                        NULL,
                        FALSE,
                        0,
                        NULL,
                        NULL,
                        &si,
                        &pi ) )
    {
        puts( "CreateProcess failed." );
        return;
    }
    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

# C6278

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6278: <variable> is allocated with array new [], but deleted with scalar delete. Destructors will not be called

This warning appears only in C++ code and indicates that the calling function has inconsistently allocated memory with the array **new []** operator, but freed it with the scalar **delete** operator. This is undefined behavior according to the C++ standard and the Microsoft Visual C++ implementation. There are at least three reasons that this is likely to cause problems:

- The constructors for the individual objects in the array are invoked, but the destructors are not invoked.
- If global, or class-specific, **operator new** and **operator delete** are not compatible with **operator new[]** and **operator delete[]**, unexpected results are likely to occur.
- It is always risky to rely on undefined behavior.

The exact ramifications of this defect are difficult to predict. It might result in leaks for classes with destructors that perform memory de-allocation; inconsistent behavior for classes with destructors that perform some semantically significant operation; or memory corruptions and crashes when operators have been overridden. In other cases the mismatch might be unimportant, depending on the implementation of the compiler and its libraries. Analysis tool cannot always distinguish between these situations.

If memory is allocated with array **new []**, it should be typically be freed with array **delete[]**.

## Example

The following sample code generates this warning:

```
class A
{
    // members
};

void f( )
{
    A *pA = new A[5];
    // code ...
    delete pA;
}
```

To correct this warning, use the following sample code:

```
void f( )
{
    A *pA = new A[5];
    // code ...
    delete[] pA;
}
```

If the underlying object in the array is a primitive type such as `int`, `float`, `enum`, or pointer, there are no destructors to be called. In these cases warning [C6283](#) is reported instead.

The use of `new` and `delete` have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6279

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6279: <variable> is allocated with scalar new, deleted with array delete []

This warning appears only in C++ code and indicates that the calling function has inconsistently allocated memory with the scalar **new** operator, but freed it with the array **delete []** operator. If memory is allocated with scalar **new**, it should typically be freed with scalar **delete**.

There are at least three reasons that this is likely to cause problems:

- The constructors for the individual objects in the array are not invoked, although the destructors are.
- If global (or class-specific) **operator new** and **operator delete** are not compatible with **operator new[]** and **operator delete[]**, unexpected results are likely to occur.

The exact ramifications of this defect are difficult to predict. It might cause random behavior or crashes due to usage of uninitialized memory because constructors are not invoked. Or, it might cause memory allocations and crashes in situations where operators have been overridden. In rare cases, the mismatch might be unimportant. Analysis tool does not currently distinguish between these situations.

## Example

The following code generates this warning:

```
class A
{
    // members
};

void f( )
{
    A *pA = new A;
    //code ...
    delete[] pA;
}
```

To correct this warning, use the following code:

```
void f( )
{
    A *pA = new A;
    //code ...
    delete pA;
}
```

To avoid these kinds of allocation problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include **shared\_ptr**, **unique\_ptr**, and **vector**. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

## See Also

- [C6014](#)

# C6280

2/21/2019 • 2 minutes to read • [Edit Online](#)

warning C6280: <variable> is allocated with <function>, but deleted with <function>

This warning indicates that the calling function has inconsistently allocated memory by using a function from one memory allocation family and freed it by using a function from another memory allocation family. The analyzer checks for this condition only when the `_Analysis_mode_(Analysis_local_leak_checks_)` SAL annotation is specified. By default, this annotation is specified for Windows kernel mode (driver) code. For more information about SAL annotations, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

For example, this warning would be produced if memory is allocated by using `malloc` but freed by using `GlobalFree` or `delete`. In the specific cases of mismatches between array `new[]` and scalar `delete`, more precise warnings are reported instead of this one.

## Example

The following sample code generates this warning.

```
// cl.exe /analyze /c /EHsc /nologo /W4
#include <sal.h>
#include <stdlib.h>

_Analysis_mode_(Analysis_local_leak_checks_)

void f(int arraySize)
{
    int *pInt = (int *)calloc(arraySize, sizeof (int));
    // code ...
    delete pInt;
}
```

To correct this warning, use this code:

```
// cl.exe /analyze /c /EHsc /nologo /W4
#include <sal.h>
#include <stdlib.h>

_Analysis_mode_(Analysis_local_leak_checks_)

void f(int arraySize)
{
    int *pInt = (int *)calloc(arraySize, sizeof (int));
    // code ...
    free(pInt);
}
```

Different API definitions can use different heaps. For example, `GlobalAlloc` uses the system heap, and `free` uses the process heap. This is likely to cause memory corruptions and crashes.

These inconsistencies apply to the `new / delete` and `malloc / free` memory allocation mechanisms. To avoid these kinds of potential inconsistencies altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers and C++ Standard Library](#).

```
// cl.exe /analyze /c /EHsc /nologo /W4
#include <sal.h>
#include <vector>
#include <memory>

using namespace std;

_Analysis_mode_(_Analysis_local_leak_checks_)

void f(int arraySize)
{
    // use unique_ptr instead of calloc/malloc/new
    unique_ptr<int[]> pInt(new int[arraySize]);

    // code ...

    // No need for free because unique_ptr
    // cleans up when out of scope.
}
```

## See Also

- [calloc](#)
- [malloc](#)
- [free](#)
- [operator new](#)
- [delete Operator](#)
- [shared\\_ptr](#)
- [unique\\_ptr](#)
- [Smart Pointers](#)

# C6281

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning 6281 - incorrect order of operations: relational operators have higher precedence than bitwise operators

This warning indicates a possible error in the operator precedence. This might produce incorrect results. You should check the precedence and use parentheses to clarify the intent. Relational operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) have higher precedence than bitwise operators (`&` | `^`).

## Example

The following code generates this warning:

```
#include <stdlib.h>
#define FORMAT 1
#define TYPE 2

void f(int input)
{
    if (FORMAT & TYPE != input)
    {
        // code...
    }
}
```

The following code uses parentheses to correct this warning:

```
#include <stdlib.h>
#define FORMAT 1
#define TYPE 2

void f(int input)
{
    if ((FORMAT & TYPE) != input)
    {
        // code...
    }
}
```

## See Also

[Compiler Warning \(level 3\) C4554](#)

# C6282

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6282: Incorrect operator: assignment of constant in Boolean context. Consider using '==' instead

This warning indicates that an assignment of a constant to a variable was detected in a test context. Assignment of a constant to a variable in a test context is almost always incorrect. Replace the `=` with `==`, or remove the assignment from the test context to resolve this warning.

## Example

The following code generates this warning:

```
void f( int i )
{
    while (i = 5)
    {
        // code
    }
}
```

To correct this warning, use the following code:

```
void f( int i )
{
    while (i == 5)
    {
        // code
    }
}
```

## See Also

[Compiler Warning \(level 4\) C4706](#)

# C6283

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6283: <variable> is allocated with array `new []`, but deleted with scalar `delete`

This warning appears only in C++ code and indicates that the calling function has inconsistently allocated memory with the array `new []` operator, but freed it with the scalar `delete` operator. This defect might cause leaks, memory corruptions, and, in situations where operators have been overridden, crashes. If memory is allocated with array `new []`, it should typically be freed with array `delete[]`.

## Example

The following code generates this warning:

```
void f( )
{
    char *str = new char[50];
    // code ...
    delete str;
}
```

To correct this warning, use the following code:

```
void f( )
{
    char *str = new char[50];
    // code ...
    delete[] str;
}
```

Warning C6283 only applies to arrays of primitive types such as, integers or characters. If elements of the array are objects of class type then warning [C6278](#) is issued.

The use of `new` and `delete` have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6284

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6284: object passed as parameter '%d' when string is required in call to <function>.

This warning indicates that the format string specifies a string, for example, a `%s` specification for `printf` or `scanf`, but a C++ object has been passed instead.

This defect might produce incorrect output or crashes.

This message is often reported due to passing a C++ object implementing some string type, for example, `std::string`, `CComBSTR` or `bstr_t`, into a C `printf`-style call. Depending on the implementation of the C++ class, that is, if the proper cast operators are defined, C++ string objects can often be used transparently whenever C strings are required; however, because parameters to `printf`-style functions are essentially untyped, no conversion to a string occurs.

Depending on the object, it might be appropriate to insert a `static_cast` operator to the appropriate string type, for example, `char *` or `TCHAR``*`, or to call a member function which returns a string, for example, `c_str()`, on instances of `std::string`.

## Example

The following code generates this warning because a `CComBSTR` is passed to the `sprintf` function:

```
#include <atbase.h>
#include <stdlib.h>

void f()
{
    char buff[50];
    CComBSTR bstrValue("Bye");

    sprintf(buff, "%ws", bstrValue);
}
```

The following code uses static cast to correct this warning:

```
#include <atbase.h>
#include <stdlib.h>

void f()
{
    char buff[50];
    CComBSTR bstrValue("Bye");

    sprintf_s(buff, 50, "%ws", static_cast<wchar_t *>(bstrValue));
}
```

## See Also

- [static\\_cast Operator](#)
- [sprintf\\_s, \\_sprintf\\_s\\_l, swprintf\\_s, \\_swprintf\\_s\\_l](#)

# C6285

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6285: (<non-zero constant> || <non-zero constant>) is always a non-zero constant. Did you intend to use the bitwise-and operator?

This warning indicates that two constant values, both greater than one, were detected as arguments to a logical-or operation that occurs in a test context. This expression is always TRUE.

Constant values greater than one suggest that the arguments to logical-or could be bit fields. Consider whether a bitwise operator might be a more appropriate operator in this case.

## Example

The following code generates this warning:

```
#include <stdio.h>
#define TESTED_VALUE 0x37
#define MASK 0xaa

void f()
{
    if (TESTED_VALUE || MASK)
    {
        puts("(TESTED_VALUE || MASK) True");
        // code ...
    }
    else
    {
        puts("(TESTED_VALUE || MASK) False");
        // code ...
    }
}
```

To correct this warning, use the following code:

```
#include <stdio.h>
#define TESTED_VALUE 0x37
#define MASK 0xaa

void f(int flag)
{
    if ((TESTED_VALUE & MASK)== flag)
    {
        puts("true");
        // code ...
    }
    else
    {
        puts("false");
        // code ...
    }
}
```

## See Also

## Expressions with Binary Operators

# C6286

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6286: (<non-zero constant> || <expression>) is always a non-zero constant. <expression> is never evaluated and may have side effects

This warning indicates that a non-zero constant was detected on the left side of a logical-or operation that occurs in a test context. The resulting expression always evaluates to TRUE. In addition, the right side of the expression appears to have side effects, and they will be lost.

This warning indicates that you may want to examine the right side of the expression carefully to ensure that any side effects needed for proper functionality are not lost.

The `(!0 || <expression>)` construction is commonly used to force execution of a controlled block.

## Example

The following code generates this warning:

```
#include <stdlib.h>
#include <stdio.h>
#define INPUT_TYPE 1

int test();

void f()
{
    if (INPUT_TYPE || test())
    {
        puts("INPUT_TYPE == 1, expression not evaluated");
        // code...
    }
    else
    {
        puts("INPUT_TYPE == 0. Call to test() returned 0");
        // code...
    }
}
```

The following code shows one possible solution by breaking `if` statement into two separate parts:

```
#include <stdlib.h>
#include <stdio.h>
#define INPUT_TYPE 1

int test();

void f()
{
    int i;
    if (INPUT_TYPE)
    {
        i = test();
        // code...
    }
    else
    {
        puts("INPUT_TYPE false");
        // code...
    }
}
```

## See Also

[Logical OR Operator: ||](#)

# C6287

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6287: redundant code: the left and right sub-expressions are identical

This warning indicates that a redundant element was detected in an expression.

It is difficult to judge the severity of this problem without examining the code. A duplicate test on its own is harmless, but the consequences of deleting the second test can be severe. The code should be inspected to ensure that a test was not omitted.

## Example

The following code generates this warning:

```
void f(int x)
{
    if ((x == 1) && (x == 1))
    {
        //logic
    }
    if ((x != 1) || (x != 1))
    {
        //logic
    }
}
```

The following code shows various ways to correct this warning:

```
void f(int x, int y)
{
    /* Remove the redundant sub-expression: */
    if (x == 1)
    {
        // logic
    }
    if (x != 1)
    {
        // logic
    }
    /* or test the missing variable: */
    if ((x == 1) && (y == 1))
    {
        // logic
    }
    if ((x != 1) || (y != 1))
    {
        // logic
    }
}
```

# C6288

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6288: Incorrect operator: mutual inclusion over `&&` is always zero. Did you intent to use `||` instead?

This warning indicates that in a test expression, a variable is being tested against two different constants and the result depends on both conditions being true. The code in these cases indicates that the programmer's intent is not captured correctly. It is important to examine the code and correct the problem; otherwise your code will not behave the way you expected it to.

This problem is generally caused by using `&&`; in place of `||`, but can also be caused by using `==` where `!=` was intended.

## Example

The following code generates this warning:

```
void f(int x)
{
    if ((x == 1) && (x == 2)) // warning
    {
        // code ...
    }
}
```

To correct this warning, use the following code:

```
void f(int x)
{
    if ((x == 1) || (x == 2))
    {
        // logic
    }

    /* or */
    if ((x != 1) && (x != 2))
    {
        // code ...
    }
}
```

The analysis tool does not warn if the expression has side effects.

# C6289

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6289: Incorrect operator: mutual exclusion over `||` is always a non-zero constant. Did you intend to use `&&` instead?

This warning indicates that in a test expression a variable is being tested against two different constants and the result depends on either condition being true. This always evaluates to true.

This problem is generally caused by using `||` in place of `&&`, but can also be caused by using `!=` where `==` was intended.

## Example

The following code generates this warning:

```
void f(int x)
{
    if ((x != 1) || (x != 3))
    {
        // code
    }
}
```

To correct this warning, use the following code:

```
void f(int x)
{
    if ((x != 1) && (x != 3))
    {
        // code
    }
}

/* or */
void f(int x)
{
    if ((x == 1) || (x == 3))
    {
        // code
    }
}
```

# C6290

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6290: Bitwise operation on logical result: ! has higher precedence than &. Use && or !(x & y) instead

This warning indicates possible confusion in the use of an operator or an operator precedence.

The `!` operator yields a Boolean result, and it has higher precedence than the `&`. The bitwise-and (`&`) operator takes two arithmetic arguments. Therefore, one of the following errors has been detected:

- The expression is mis-parenthesised:

Because the result of `!` is Boolean (zero or one), an attempt to test that two variables have bits in common will only end up testing that the lowest bit is present in the right side: `((!8) & 1) == 0`.

- The `!` operator is incorrect, and should be a `~` instead:

The `!` operator has a Boolean result, while the `~` operator has an arithmetic result. These operators are never interchangeable, even when operating on a Boolean value (zero or one): `((!0x01) & 0x10) == 0x0`, while `((~0x01) & 0x10) == 0x10`.

- The binary operator `&` is incorrect, and should instead be `&&`:

While `&` can sometimes be interchanged with `&&`, it is not equivalent because it forces evaluation of the right side of the expression. Certain side effects in this type of expression can be terminal.

It is difficult to judge the severity of this problem without examining the code. The code should be inspected to ensure that the intended test is occurring.

## Example

The following code generates this warning:

```
void f(int x, int y)
{
    if (!x & y)
    {
        // code ...
    }
}
```

To correct this warning, use the following sample code:

```
void f(int x, int y)
{
    /* When testing that x has no bits in common with y. */
    if (!(x & y))
    {
        // code
    }

    /* When testing for the complement of x in y. */
    if ((~x) & y)
    {
        // code ...
    }
}

#include <windows.h>
void fC(int x, BOOL y )
{
    /* When y is a Boolean or Boolean result. */
    if ((!x) && y)
    {
        // code ...
    }
}
```

Warning C6317 is reported if the `!` operator is on the right side of the `&` operator.

## See Also

- [C6317](#)

# C6291

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6291: Bitwise operation on logical result: `!` has higher precedence than `|`. Use `||` or `(!(x | y))` instead

The `!` operator yields a Boolean result, and the `|` (bitwise-or) operator takes two arithmetic arguments. The `!` operator also has higher precedence than `|`.

Therefore, one of the following errors has been detected:

- The expression is mis-parenthesised:

Because the result of `!` is Boolean (zero or one), an attempt to test that two variables have bits set will only end up testing that the lowest bit is present in the right side: `((!x) | y) != (!(x | y))` when `x == 0` and `y == 1`.

- The `!` operator is incorrect, and should be a `~` instead:

The `!` operator has a Boolean result, but the `~` operator has an arithmetic result. These operators are never interchangeable, even when operating on a Boolean value (zero or one): `((!x) | y) != ((~x) | y)` when `x == 1` and `y == 0`.

- The binary operator `|` is incorrect, and should instead be `||`:

Even though `|` can sometimes be interchanged with `||`, it is not equivalent because it forces evaluation of the right side of the expression. Certain side-effects in this type of expression can be terminal:

`(!p | (*p == '\0'))`, when `p == NULL`, we must dereference it to evaluate the other half of the expression.

This warning is not reported if the `!` operator is on the right side of the `|` operator because this case is typically just the relatively harmless case of an incorrect operator.

It is difficult to judge the severity of this problem without examining the code. The code should be inspected to ensure that the intended test is occurring.

This warning always indicates possible confusion in the use of an operator or operator precedence.

## Example

The following code generates this warning:

```
void f(int x, int y )
{
    if (!x | y)
    {
        //code
    }
}
```

To correct this warning, use one of the examples shown in the following code:

```
void fC(int x, int y )
{
    /* When checking whether any bits are set in either x or y. */
    if (!(x | y))
    {
        // code
    }
    /* When checking whether bits are set in either */
    /* the complement of x or in y. */
    if ((~x) | y)
    {
        // code
    }
}

#include <windows.h>
void f(int x, BOOL y )
{
    /* When y is a Boolean or Boolean result. */
    if ((!x) || y)
    {
        // code
    }
}
```

# C6292

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6292: ill-defined for-loop: counts up from maximum

This warning indicates that a for-loop might not function as intended.

It occurs when a loop counts up from a maximum, but has a lower termination condition. This loop will terminate only after integer overflow occurs.

## Example

The following code generates this warning:

```
void f( )
{
    int i;

    for (i = 100; i >= 0; i++)
    {
        // code ...
    }
}
```

To correct this warning, use the following code:

```
void f( )
{
    int i;

    for (i = 100; i >= 0; i--)
    {
        // code ...
    }
}
```

# C6293

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6293: Ill-defined for-loop: counts down from minimum

This warning indicates that a for-loop might not function as intended. It occurs when a loop counts down from a minimum, but has a higher termination condition.

A signed—or unsigned—index variable together with a negative increment will cause the loop to count negative until an overflow occurs. This will terminate the loop.

## Example

The following sample code generates this warning:

```
void f( )
{
    signed char i;

    for (i = 0; i < 100; i--)
    {
        // code ...
    }
}
```

To correct this warning, use the following code:

```
void f( )
{
    signed char i;

    for (i = 0; i < 100; i++)
    {
        // code ...
    }
}
```

# C6294

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6294: Ill-defined for-loop: initial condition does not satisfy test. Loop body not executed

This warning indicates that a for-loop cannot be executed because the terminating condition is true. This warning suggests that the programmer's intent is not correctly captured.

## Example

The following sample code generates this warning because MAX\_VALUE is 0:

```
#define MAX_VALUE 0
void f()
{
    int i;
    for (i = 0; i < MAX_VALUE; i++)
    {
        // code
    }
}
```

The following sample code corrects this warning by changing the value of MAX\_VALUE to 25

```
#define MAX_VALUE 25
void f()
{
    int i;
    for (i = 0; i < MAX_VALUE; i++)
    {
        // code
    }
}
```

# C6295

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6295: Ill-defined for-loop: <variable> values are of the range "min" to "max". Loop executed indefinitely

This warning indicates that a for-loop might not function as intended. The for-loop tests an unsigned value against zero (0) with  $\geq$ . The result is always true, therefore the loop is infinite.

## Example

The following code generates this warning:

```
void f( )
{
    for (unsigned int i = 100; i >= 0; i--)
    {
        // code ...
    }
}
```

To correct this warning, use the following code:

```
void f( )
{
    for (unsigned int i = 100; i > 0; i--)
    {
        // code ...
    }
}
```

# C6296

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6296: Ill-defined for-loop: Loop body only executed once

This warning indicates that a for-loop might not function as intended. When the index is unsigned and a loop counts down from zero, its body is run only once.

## Example

The following code generates this warning:

```
void f( )
{
    unsigned int i;

    for (i = 0; i < 100; i--)
    {
        // code ...
    }
}
```

To correct this warning, use the following code:

```
void f( )
{
    unsigned int i;

    for (i = 0; i < 100; i++)
    {
        // code ...
    }
}
```

# C6297

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6297: Arithmetic overflow: 32-bit value is shifted, then cast to 64-bit value. Result may not be an expected value

This warning indicates incorrect behavior that results from integral promotion rules and types larger than those in which arithmetic is typically performed.

In this case, a 32-bit value was shifted left, and the result of that shift was cast to a 64-bit value. If the shift overflowed the 32-bit value, bits are lost.

If you do not want to lose bits, cast the value to be shifted to a 64-bit quantity before it is shifted. If you want to lose bits, performing the appropriate cast to unsigned long or a short type, or masking the result of the shift will eliminate this warning and make the intent of the code more clear.

## Example

The following code generates this warning:

```
void f(int i)
{
    unsigned __int64 x;

    x = i << 34;
    // code
}
```

To correct this warning, use the following code:

```
void f(int i)
{
    unsigned __int64 x;
    // code
    x = ((unsigned __int64)i) << 34;
}
```

## See Also

[Compiler Warning \(level 1\) C4293](#)

# C6298

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6298: using a read-only string <pointer> as a writable string argument: this will attempt to write into static read-only memory and cause random crashes

This warning indicates the use of a constant string as an argument to a function that might modify the contents of that string. Because the compiler allocates constant strings in a static read-only memory, any attempts to modify it cause access violations and random crashes.

This can be avoided by storing the constant string into a local array and then using the array as the argument to the function.

## Example

The following sample code generates this warning:

```
#include <windows.h>
#include <stdio.h>

void f()
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof( si ) );
    si.cb = sizeof( si );
    ZeroMemory( &pi, sizeof( pi ) );
    if( !CreateProcess(NULL,
                      "\c:\\Windows\\system32\\calc.exe\",
                      NULL,
                      NULL,
                      FALSE,
                      0,
                      NULL,
                      NULL,
                      &si,
                      &pi ) )
    {
        puts( "CreateProcess failed." );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

To correct this warning, use the following sample code:

```
#include <windows.h>
#include <stdio.h>

void f( )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    char szCmdLine[] = "c:\\Windows\\system32\\calc.exe\"";
    ZeroMemory( &si, sizeof( si ) );
    si.cb = sizeof( si );
    ZeroMemory( &pi, sizeof( pi ) );

    if( !CreateProcess(NULL,
                      szCmdLine,
                      NULL,
                      NULL,
                      FALSE,
                      0,
                      NULL,
                      NULL,
                      &si,
                      &pi ) )
    {
        puts( "CreateProcess failed." );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

# C6299

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6299: explicitly comparing a bit field to a Boolean type will yield unexpected results

This warning indicates an incorrect assumption that Booleans and bit fields are equivalent. Assigning 1 to bit fields will place 1 in its single bit; however, any comparison of this bit field to 1 includes an implicit cast of the bit field to a signed int. This cast will convert the stored 1 to a -1 and the comparison can yield unexpected results.

## Example

The following code generates this warning:

```
struct myBits
{
    short flag : 1;
    short done : 1;
    //other members
} bitType;

void f( )
{
    if (bitType.flag == 1)
    {
        // code...
    }
}
```

To correct this warning, use a bit field as shown in the following code:

```
void f ()
{
    if(bitType.flag==bitType.done)
    {
        // code...
    }
}
```

# C6302

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6302: format string mismatch: character string passed as parameter <number> when wide character string is required in call to <function>

This warning indicates that the format string specifies that a wide character string is required. However, a character string is being passed. This defect is likely to cause a crash or a corruption of some form.

## Example

The following sample code generates this warning because a character string is passed to `wprintf` function:

```
#include<stdio.h>

void f()
{
    char buff[5] = "hi";

    wprintf(L"%s", buff);
}
```

The following sample code uses `%hs` to specify a single-byte character string with `wprintf` function:

```
#include<stdio.h>

void f()
{
    char buff[5] = "hi";

    wprintf(L"%hs", buff);
}
```

The following sample code uses safe string manipulation function `wprintf_s` to correct this warning:

```
#include<stdio.h>

void f()
{
    char buff[5] = "hi";

    wprintf_s(L"%hs", buff);
}
```

# C6303

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6303: format string mismatch: wide character string passed as parameter <number> when character string is required in call to <function>

This warning indicates that the format string specifies that a character string is required. However, a wide character string is being passed. This defect is likely to cause a crash or corruption of some form.

## Example

The following sample code generates this warning:

```
#include <stdio.h>

void f()
{
    wchar_t buff[5] = L"hi";

    printf("%s", buff);
}
```

To correct this warning, use `%ls` as shown in the following sample code:

```
#include <stdio.h>

void f()
{
    wchar_t buff[5] = L"hi";

    printf("%ls", buff);
}
```

The following sample code uses safe string manipulation function `printf_s` to correct this warning:

```
#include <stdio.h>

void f()
{
    wchar_t buff[5] = L"hi";

    printf_s("%ls",buff);
}
```

# C6305

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6305: potential mismatch between sizeof and countof quantities

This warning indicates that a variable holding a `sizeof` result is being added to or subtracted from a pointer or `countof` expression. This will cause unexpected scaling in pointer arithmetic.

## Example

The following code generates this warning:

```
void f(int *p)
{
    int cb=sizeof(int);
    //code...
    p +=cb; // warning 6305
}
```

To correct this warning, use the following code:

```
void f(int *p)
{
    // code...
    p += 1;
}
```

# C6306

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6306: incorrect call to <function>: consider using <function> which accepts a va\_list as an argument

This warning indicates an incorrect function call. The `printf` family includes several functions that take a variable list of arguments; however, these functions cannot be called with a `va_list` argument. There is a corresponding `vprintf` family of functions that can be used for such calls. Calling the wrong print function will cause incorrect output.

## Example

The following code generates this warning:

```
#include <stdio.h>
#include <stdarg.h>

void f(int i, ...)
{
    va_list v;

    va_start(v, i);
    //code...
    printf("%s", v); // warning 6306
    va_end(v);
}
```

To correct this warning, use the following code:

```
#include <stdio.h>
#include <stdarg.h>

void f(int i, ...)
{
    va_list v;

    va_start(v, i);
    //code...
    vprintf_s("%d",v);
    va_end(v);
}
```

## See Also

[C6273](#)

# C6308

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6308: 'realloc' may return null pointer: assigning a null pointer to <variable>, which is passed as an argument to 'realloc', will cause the original memory block to be leaked

This warning indicates a memory leak that is the result of the incorrect use of a reallocation function. Heap reallocation functions do not free the passed buffer if reallocation is unsuccessful. To correct the defect, assign the result of the reallocation function to a temporary, and then replace the original pointer after successful reallocation.

## Example

The following sample code generates this warning:

```
#include <malloc.h>
#include <windows.h>

void f( )
{
    char *x;
    x = (char *) malloc(10);
    if (x != NULL)
    {
        x = (char *) realloc(x, 512);
        // code...
        free(x);
    }
}
```

To correct this warning, use the following code:

```
#include <malloc.h>
#include <windows.h>

void f()
{
    char *x, *tmp;

    x = (char *) malloc(10);

    if (x != NULL)
    {
        tmp = (char *) realloc(x, 512);
        if (tmp != NULL)
        {
            x = tmp;
        }
        free(x);
    }
}
```

This warning might generate noise if there is a live alias to the buffer-to-be-reallocated at the time of the assignment of the result of the reallocation function.

To avoid these kinds of problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and

[C++ Standard Library](#).

## See Also

[C6014](#)

# C6310

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6310: illegal constant in exception filter can cause unexpected behavior

This message indicates that an illegal constant was detected in the filter expression of a structured exception handler. The constants defined for use in the filter expression of a structured exception handler are:

- `EXCEPTION_CONTINUE_EXECUTION`
- `EXCEPTION_CONTINUE_SEARCH`
- `EXCEPTION_EXECUTE_HANDLER`

These values are defined in the runtime header file `excpt.h`.

Using a constant that is not in the preceding list can cause unexpected behavior.

## Example

The following code generates this warning:

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

BOOL LimitExceeded();

void fd( )
{
    __try
    {
        if (LimitExceeded())
        {
            RaiseException(EXCEPTION_ACCESS_VIOLATION,0,0,0);
        }
        else
        {
            // code
        }
    }
    __except (EXCEPTION_ACCESS_VIOLATION)
    {
        puts("Exception Occurred");
    }
}
```

To correct this warning, use the following code:

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

BOOL LimitExceeded();

void fd( )
{
    __try
    {
        if (LimitExceeded())
        {
            RaiseException(EXCEPTION_ACCESS_VIOLATION,0,0,0);
        }
        else
        {
            // code
        }
    }
    __except (GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
              EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        puts("Exception Occurred");
    }
}
```

# C6312

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6312: Possible infinite loop: use of the constant EXCEPTION\_CONTINUE\_EXECUTION in the exception-filter expression of a try-except

This warning indicates the use of the constant `EXCEPTION_CONTINUE_EXECUTION` (or another constant that evaluates to -1) in the filter expression of a structured exception handler. Use of the constant value

`EXCEPTION_CONTINUE_EXECUTION` could lead to an infinite loop. For example, if an exception was raised by hardware, the instruction that caused the exception will be restarted. If the address that caused the exception is still bad, another exception will occur and be handled in the same way. This causes an infinite loop.

An explicit call to `RaiseException` will not directly cause an infinite loop, but it will continue execution of the code in the protected block. This can be unexpected, and could lead to an infinite loop if `RaiseException` was used to avoid dereferencing an invalid pointer.

Typically, `EXCEPTION_CONTINUE_EXECUTION` should be returned only by a function called in the filter expression, which has a chance to fix either the pointer that caused the exception or the underlying memory.

## Example

The following code generates this warning:

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

void f (char *ptr)
{
    __try
    {
        // exception occurs if the caller passes null ptr
        // code...
        *ptr = '\0';
    }
    __except (EXCEPTION_CONTINUE_EXECUTION)
    // When EXCEPTION_CONTINUE_EXECUTION is used, the handler
    // block of the structured exception handler is not executed.
    {
        puts("This block is never executed");
    }
}
```

To correct this warning, use the following code:

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

void f (char *ptr)
{
    __try
    {
        // exception occurs if the caller passes null ptr
        // code...
        *ptr = '\0';
    }
    __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
              EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        puts("Error Occurred");
    }
}
```

# C6313

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6313: Incorrect operator: Zero-valued flag cannot be tested with bitwise-and. Use an equality test to check for zero-valued flags

This warning indicates that a constant value of zero was provided as an argument to the bitwise-and (`&`) operator in a test context. The resulting expression is constant and evaluates to false; the result is different than intended.

This is typically caused by using bitwise-and to test for a flag that has the value zero. To test zero-valued flags, a test for equality must be performed, for example, using `==` or `!=`.

## Example

The following code generates this warning:

```
#define FLAG 0

void f(int Flags )
{
    if (Flags & FLAG)
    {
        // code
    }
}
```

To correct this warning, use the following code:

```
#define FLAG 0

void f(int Flags )
{
    if (Flags == FLAG)
    {
        // code
    }
}
```

# C6314

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6314: Incorrect order of operations: bitwise-or has higher precedence than the conditional-expression operator. Add parentheses to clarify intent

This message indicates that an expression that contains a bitwise-or operator (`|`) was detected in the tested expression of a conditional operation (`? :`).

The conditional operator has lower precedence than bitwise operators. If the tested expression should contain the bitwise-or operator, then parentheses should be added around the conditional-expression.

## Example

The following code generates this warning:

```
int SystemState();

int f(int SignalValue)
{
    return SystemState() | (SignalValue != 0) ? 1 : 0;
}
```

To correct this warning, use the following code:

```
int SystemState();

int f(int SignalValue)
{
    return SystemState() | ((SignalValue != 0) ? 1 : 0);
}
```

## See Also

- [Bitwise Inclusive OR Operator: |](#)
- [Conditional Operator: ? :](#)

# C6315

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6315: Incorrect order of operations: bitwise-and has higher precedence than bitwise-or. Add parentheses to clarify intent

This warning indicates that an expression in a test context contains both bitwise-and (`&`) and bitwise-or (`|`) operations, but causes a constant because the bitwise-or operation happens last. Parentheses should be added to clarify intent.

## Example

The following code generates this warning:

```
void f( int i )
{
    if ( i & 2 | 4 ) // warning
    {
        // code
    }
}
```

To correct this warning, add parenthesis as shown in the following code:

```
void f( int i )
{
    if ( i & ( 2 | 4 ) )
    {
        // code
    }
}
```

# C6316

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6316: Incorrect operator: tested expression is constant and non-zero. Use bitwise-and to determine whether bits are set

This warning indicates the use of bitwise-or (`|`) when bitwise-and (`&`) should have been used. Bitwise-or adds bits to the resulting expression, whereas bitwise-and selects only those bits in common between its two operators. Tests for flags must be performed with bitwise-and or a test of equality.

## Example

The following code generates this warning:

```
#define INPUT_VALUE 2
void f( int Flags)
{
    if (Flags | INPUT_VALUE) // warning
    {
        // code
    }
}
```

To correct this warning, use the following code:

```
#define ALLOWED 1
#define INPUT_VALUE 2

void f( int Flags)
{
    if ((Flags & INPUT_VALUE) == ALLOWED)
    {
        // code
    }
}
```

# C6317

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6317: incorrect operator: logical-not (!) is not interchangeable with ones-complement (~)

This warning indicates that a logical-not (`!`) is being applied to a constant that is likely to be a bit-flag. The result of logical-not is Boolean; it is incorrect to apply the bitwise-and (`&`) operator to a Boolean value. Use the ones-complement (`~`) operator to flip flags.

## Example

The following code generates this warning:

```
#define FLAGS 0x4004

void f(int i)
{
    if (i & !FLAGS) // warning
    {
        // code
    }
}
```

To correct this warning, use the following code:

```
#define FLAGS 0x4004

void f(int i)
{
    if (i & ~FLAGS)
    {
        // code
    }
}
```

## See Also

- [Bitwise AND Operator: &](#)
- [Logical Negation Operator: !](#)
- [One's Complement Operator: ~](#)

# C6318

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6318: Ill-defined \_\_try/\_\_except: use of the constant EXCEPTION\_CONTINUE\_SEARCH or another constant that evaluates to zero in the exception-filter expression. The code in the exception handler block is not executed

This warning indicates that if an exception occurs in the protected block of this structured exception handler, the exception will not be handled because the constant `EXCEPTION_CONTINUE_SEARCH` is used in the exception filter expression.

This code is equivalent to the protected block without the exception handler block because the handler block is not executed.

## Example

The following code generates this warning:

```
#include <excpt.h>
#include <stdio.h>

void f (char *pch)
{
    __try
    {
        // assignment might fail
        *pch = 0;
    }
    __except (EXCEPTION_CONTINUE_SEARCH) // warning 6318
    {
        puts("Exception Occurred");
    }
}
```

To correct this warning, use the following code:

```
#include <excpt.h>
#include <stdio.h>
#include <windows.h>

void f (char *pch)
{
    __try
    {
        // assignment might fail
        *pch = 0;
    }
    __except( (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) ?
              EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH )
    {
        puts("Access violation");
    }
}
```

## See Also

## try-except Statement

# C6319

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6319: use of the comma-operator in a tested expression causes the left argument to be ignored when it has no side-effects

This warning indicates an ignored sub-expression in test context because of the comma-operator (,). The comma operator has left-to-right associativity. The result of the comma-operator is the last expression evaluated. If the left expression to comma-operator has no side effects, the compiler might omit code generation for the expression.

## Example

The following code generates this warning:

```
void f()
{
    int i;
    int x[10];

    // code
    for ( i = 0; x[i] != 0, x[i] < 42; i++) // warning
    {
        // code
    }
}
```

To correct this warning, use the logical AND operator as shown in the following code:

```
void f()
{
    int i;
    int x[10];

    // code

    for ( i = 0; (x[i] != 0) && (x[i] < 42); i++)
    {
        // code
    }
}
```

## See Also

- [Logical AND Operator: &&](#)
- [Comma Operator:,](#)

# C6320

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6320: exception-filter expression is the constant EXCEPTION\_EXECUTE\_HANDLER. This may mask exceptions that were not intended to be handled

This warning indicates the side effect of using EXCEPTION\_EXECUTE\_HANDLER constant in \_\_except block. In this case, the statement in the \_\_except block will always execute to handle the exception, including exceptions you did not want to handle in a particular function. It is recommended that you check the exception before handling it.

## Example

The following code generates this warning because the \_\_except block will try to handle exceptions of all types:

```
#include <stdio.h>
#include <excpt.h>

void f(int *p)
{
    __try
    {
        puts("in try");
        *p = 13; // might cause access violation exception
        // code ...
    }
    __except(EXCEPTION_EXECUTE_HANDLER) // warning
    {
        puts("in except");
        // code ...
    }
}
```

To correct this warning, use `GetExceptionCode` to check for a particular exception before handling it as shown in the following code:

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

void f(int *p)
{
    __try
    {
        puts("in try");
        *p = 13; // might cause access violation exception
        // code ...
    }
    __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        puts("in except");
        // code ...
    }
}
```

## See Also

[try-except Statement](#)

# C6322

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6322: empty \_except block

This message indicates that there is no code in the \_except block. As a result, exceptions might go unhandled.

## Example

The following code generates this warning:

```
#include <stdio.h>
#include <excpt.h>
#include <windows.h>

void fd(char *pch)
{
    __try
    {
        // exception rasied if pch is null
        *pch= 0 ;
    }
    __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION)
    {
    }
}
```

To correct this warning, use the following code:

```
#include <stdio.h>
#include <excpt.h>
#include <windows.h>

void f(char *pch)
{
    __try
    {
        // exception rasied if pch is null
        *pch= 0 ;
    }
    __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        // code to handle exception
        puts("Exception Occurred");
    }
}
```

## See Also

[try-except Statement](#)

# C6323

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6323 - use of arithmetic operator on Boolean type(s)

This warning occurs if arithmetic operators are used on Boolean data types. Use of incorrect operator might yield incorrect results. It also indicates that the programmer's intent is not reflected in the code.

## Example

The following code generates this warning:

```
int test(bool a, bool b)
{
    int c = a + b;      //C6323
    return c;
}
```

To correct this warning, use correct data type and operator.

```
int test(int a, int b)
{
    int c = a + b;
    return c;
}
```

# C6324

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6324: potential incorrect use of <function1>: Did you intend to use <function2>?

This warning indicates that a string copy function was used where a string comparison function should have been used. Incorrect use of function can cause an unexpected logic error.

## Example

The following code generates this warning:

```
#include <string.h>

void f(char *title )
{
    if (strcpy (title, "Manager") == 0) // warning 6324
    {
        // code
    }
}
```

To correct this warning, use `strcmp` as shown in the following code:

```
#include <string.h>

void f(char *title )
{
    if (strcmp (title, "Manager") == 0)
    {
        // code
    }
}
```

## See Also

- [strcpy, wcscpy, \\_mbscpy](#)
- [strcpy\\_s, wcscpy\\_s, \\_mbscpy\\_s](#)
- [strncpy, \\_strncpy\\_l, wcsncpy, \\_wcsncpy\\_l, \\_mbsncpy, \\_mbsncpy\\_l](#)
- [\\_mbsnbcpy, \\_mbsnbcpy\\_l](#)
- [strcmp, wcscmp, \\_mbscmp](#)
- [strncmp, wcsncmp, \\_mbsncmp, \\_mbsncmp\\_l](#)
- [\\_mbsnbcmp, \\_mbsnbcmp\\_l](#)

# C6326

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6326: potential comparison of a constant with another constant

This warning indicates a potential comparison of a constant with another constant, which is redundant code. You must check to make sure that your intent is properly captured in the code. In some cases, you can simplify the test condition to achieve the same result.

## Example

The following code generates this warning because two constants are compared:

```
#define LEVEL
const int STD_LEVEL = 5;

const int value =
#endif LEVEL
10;
#else
5;
#endif

void f()
{
    if( value > STD_LEVEL)
    {
        // code...
    }
    else
    {
        // code...
    }
}
```

The following code shows one way to correct this warning by using the #ifdef statements to determine which code should execute:

```
#define LEVEL
const int STD_LEVEL = 5;

const int value =
#endif LEVEL
10;
#else
5;
#endif

void f ()
{
#ifndef LEVEL
{
    // code...
}
#else
{
    // code...
}
#endif
}
```

# C6328

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6328: Size mismatch: <type> passed as parameter <number> when <type> is required in call to <function>

For C runtime character-based routines in the family name `is xxx()`, passing an argument of type `char` can have unpredictable results. For example, an SBCS or MBCS single-byte character of type `char` with a value greater than `0x7F` is a negative value. If a `char` is passed, the compiler might convert the value to a signed `int` or a signed `long`. This value could be sign-extended by the compiler, with unexpected results. For example, `isspace` accepts an argument of type `int`; however, the valid range of values for its input argument is:

`0 <= c <= 255`, plus the special value `EOF`.

## Example

By default, `char` is a signed type in Visual C++, so the range of values of a variable of type `char` is `-128 <= c <= 127`. Therefore, if you did the following:

```
#include <iostream>

void f( )
{
    char c = -37;
    int retVal = isspace( c );
    // code ...
}
```

`c` would be sign-extended to a signed `int` with a value of -37, which is outside the valid range for `isspace`.

To correct this problem, you can use `static_cast`, as shown in the following code:

```
#include <iostream>

void f( )
{
    char c = -37;
    int retVal = isspace( static_cast<unsigned char>(c) );
    // code ...
}
```

Warning C6328 exists specifically to catch this bug. For characters in the 7-bit ASCII range the cast is unnecessary, but characters outside that range can have unpredictable results, up to and including program fault and termination.

# C6329

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6329: Return value for a call to <function> should not be checked against <number>

The program is comparing a number against the return value from a call to `CreateFile`. If `CreateFile` succeeds, it returns an open handle to the object. If it fails, it returns `INVALID_HANDLE_VALUE`.

## Example

This code could cause the warning:

```
if (CreateFile() == NULL)
{
    return;
}
```

## Example

This code corrects the error:

```
if (CreateFile() == INVALID_HANDLE_VALUE)
{
    return;
}
```

# C6330

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6330: Incorrect type passed as parameter in call to function

# C6331

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6331: Invalid parameter: passing MEM\_RELEASE and MEM\_DECOMMIT in conjunction to <function> is not allowed. This results in the failure of this call

This message indicates that an invalid parameter being passed to VirtualFree or VirtualFreeEx. VirtualFree and VirtualFreeEx both reject the flags (MEM\_RELEASE | MEM\_DECOMMIT) in combination. Therefore, the values MEM\_DECOMMIT and MEM\_RELEASE may not be used together in the same call.

It is not required for decommit and release to occur as independent steps. Releasing committed memory will decommit the pages as well. Also, ensure the return value of this function is not ignored.

## Example

The following sample code generates this warning:

```
#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID fd( VOID )
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc (
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS );

    if (lpvBase)
    {
        // code to access memory
    }
    else
    {
        return;
    }
    bSuccess = VirtualFree(lpvBase,
        0,
        MEM_DECOMMIT | MEM_RELEASE); // warning
    // code...
}
```

To correct this warning, do not pass MEM\_DECOMMIT value to VirtualFree call as shown in the following code:

```

#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( VOID )
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc (
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS );
    if (lpvBase)
    {
        // code to access memory
    }
    else
    {
        return;
    }
    bSuccess = VirtualFree(lpvBase, 0, MEM_RELEASE);
    // code...
}

```

Note that the use of malloc and free (and related dynamic memory allocation APIs) have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

## See Also

- [VirtualAlloc Method](#)
- [VirtualFree Method](#)

# C6332

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6332: Invalid parameter: passing zero as the dwFreeType parameter to <function> is not allowed. This results in the failure of this call

This warning indicates that an invalid parameter is being passed to VirtualFree or VirtualFreeEx. VirtualFree and VirtualFreeEx both reject a dwFreeType parameter of zero. The dwFreeType parameter can be either MEM\_DECOMMIT or MEM\_RELEASE. However, the values MEM\_DECOMMIT and MEM\_RELEASE may not be used together in the same call. Also, make sure that the return value of the VirtualFree function is not ignored.

## Example

The following code generates this warning because an invalid parameter is passed to the VirtualFree function:

```
#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( VOID )
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS );

    if (lpvBase)
    {
        // code to access memory
    }
    else
    {
        return;
    }

    bSuccess = VirtualFree( lpvBase, 0, 0 );
    // code ...
}
```

To correct this warning, modify the call to the VirtualFree function as shown in the following code:

```

#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( VOID )
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS );
    if (lpvBase)
    {
        // code to access memory
    }
    else
    {
        return;
    }

    bSuccess = VirtualFree( lpvBase, 0, MEM_RELEASE );
    // code ...
}

```

The use of `VirtualAlloc` and `VirtualFree` have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include `shared_ptr`, `unique_ptr`, and `vector`. For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

## See Also

- [VirtualAlloc Method](#)
- [VirtualFree Method](#)

# C6333

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6333: Invalid parameter: passing MEM\_RELEASE and a non-zero dwSize parameter to <function> is not allowed. This results in the failure of this call

This warning indicates an invalid parameter is being passed to VirtualFree or VirtualFreeEx. Both of these functions reject a dwFreeType of MEM\_RELEASE with a non-zero value of dwSize. When passing MEM\_RELEASE, the dwSize parameter must be zero. Also, make sure that the return value of this function is not ignored.

## Example

The following sample code generates this warning:

```
#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( VOID )
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS );

    if (lpvBase)
    {
        // code to access memory
    }
    else
    {
        return;
    }

    bSuccess = VirtualFree(lpvBase, PAGELIMIT * dwPageSize, MEM_RELEASE);
    //code...
}
```

To correct this warning, use the following sample code:

```

#include <windows.h>
#define PAGELIMIT 80

DWORD dwPages = 0; // count of pages
DWORD dwPageSize; // page size

VOID f( VOID )
{
    LPVOID lpvBase; // base address of the test memory
    BOOL bSuccess;
    SYSTEM_INFO sSysInfo; // system information

    GetSystemInfo( &sSysInfo );
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the process's virtual address space
    lpvBase = VirtualAlloc(
        NULL, // system selects address
        PAGELIMIT*dwPageSize, // size of allocation
        MEM_RESERVE,
        PAGE_NOACCESS );

    if (lpvBase)
    {
        // code to access memory
    }
    else
    {
        return;
    }
    bSuccess = VirtualFree(lpvBase, 0, MEM_RELEASE );

    // VirtualFree(lpvBase, PAGELIMIT * dwPageSize, MEM_DECOMMIT);
    // code...
}

```

You can also use `VirtualFree(lpvBase, PAGELIMIT * dwPageSize, MEM_DECOMMIT)`; call to decommit pages, and later release them using `MEM_RELEASE` flag.

## See Also

- [VirtualAlloc Method](#)
- [VirtualFree Method](#)

# C6334

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6334: sizeof operator applied to an expression with an operator may yield unexpected results

This warning indicates a misuse of the `sizeof` operator. The `sizeof` operator, when applied to an expression, yields the size of the type of the resulting expression.

For example, in the following code:

```
char      a[10];
size_t   x;

x = sizeof (a - 1);
```

`x` will be assigned the value 4, not 9, because the resulting expression is no longer a pointer to the array `a`, but simply a pointer.

## Example

The following code generates this warning:

```
void f( )
{
    size_t x;
    char a[10];

    x= sizeof (a - 4);
    // code...
}
```

To correct this warning, use the following code:

```
void f( )
{
    size_t x;
    char a[10];

    x= sizeof (a) - 4;
    // code...
}
```

## See Also

[sizeof Operator](#)

# C6335

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6335: leaking process information handle <handlename>

This warning indicates that the process information handles returned by the CreateProcess family of functions need to be closed using CloseHandle. Failure to do so will cause handle leaks.

## Example

The following code generates this warning:

```
#include <windows.h>
#include <stdio.h>

void f( )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( "C:\\WINDOWS\\system32\\calc.exe",
                        NULL,
                        NULL,
                        NULL,
                        FALSE,
                        0,
                        NULL,
                        NULL,
                        &si,      // Pointer to STARTUPINFO structure.
                        &pi ) ) // Pointer to PROCESS_INFORMATION
    {
        puts("Error");
        return;
    }
    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );
    CloseHandle( pi.hProcess );
}
```

To correct this warning, call `CloseHandle( pi.hThread )` to close thread handle as shown in the following code:

```
#include <windows.h>
#include <stdio.h>

void f( )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( "C:\\\\WINDOWS\\\\system32\\\\calc.exe",
                        NULL,
                        NULL,
                        NULL,
                        FALSE,
                        0,
                        NULL,
                        NULL,
                        &si,      // Pointer to STARTUPINFO structure.
                        &pi ) ) // Pointer to PROCESS_INFORMATION
    {
        puts("Error");
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

For more information, see [CreateProcess Function](#) and [CloseHandle Function](#).

# C6336

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6336: arithmetic operator has precedence over question operator, use parentheses to clarify intent

This warning indicates a possible operator precedence problem. The '+','-','\*' and '/' operators have precedence over the '?' operator. If the precedence in the expression is not correct, use parentheses to change the operator precedence.

## Example

The following code generates this warning:

```
int Count();  
  
void f(int flag)  
{  
    int result;  
    result = Count() + flag ? 1 : 2;  
    // code...  
}
```

To correct this warning, add parenthesis as shown in the following code:

```
int Count();  
  
void f(int flag)  
{  
    int result;  
    result = Count() + (flag ? 1 : 2);  
    // code...  
}
```

## See Also

[C++ Built-in Operators, Precedence and Associativity](#)

# C6340

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6340: Mismatch on sign: Incorrect type passed as parameter in call to function

# C6381

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6381: Shutdown API <function> requires a valid dwReason or lpMessage

This warning is issued if `InitiateSystemShutdownEx` is called:

- Without passing a valid shutdown reason (`dwReason`). If `dwReason` parameter is zero, the default is an undefined shutdown. By default, it is also an unplanned shutdown. You should use one of the System Shutdown Reason Codes for this parameter.
- Without passing a shutdown message (`lpMessage`).

We recommend that you use appropriate parameters when calling this API to help system administrators determine the cause of the shutdown.

## Example

The following code generates this warning because `dwReason` is zero and `lpMessage` is null:

```
void f()
{
    //...
    BOOL bRet;
    bRet = InitiateSystemShutdownEx( NULL,
                                    NULL, // message
                                    0,
                                    FALSE,
                                    TRUE,
                                    0); // shutdown reason
    // ...
}
```

To correct this warning, specify `dwReason` and `lpMessage` as shown in the following code:

```
#include <windows.h>
void f()
{
    //...
    BOOL bRet;
    bRet = InitiateSystemShutdownEx( NULL,
                                    "Hardware Failure", // message
                                    0,
                                    FALSE,
                                    TRUE,
                                    SHTDN_REASON_MAJOR_HARDWARE ); // reason
    // ...
}
```

# C6383

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6383: buffer overrun due to conversion of an element count into a byte count: an element count is expected for parameter <number> in call to <function>

This warning indicates that a non-constant byte count is being passed when an element count is required.

Typically, this occurs when a variable is multiplied by the `sizeof` a type, but code analysis suggests that an element count is required.

## Example

The following code generates this warning:

```
#include <string.h>

void f( wchar_t* t, wchar_t* s, int n )
{
    // code...
    wcsncpy (t, s, n*sizeof(wchar_t)); // warning 6383
    // code ...
}
```

To correct this warning, do not multiply the variable with the `sizeof` a type as shown in the following code:

```
void f( wchar_t* t, wchar_t* s, int n )
{
    // code
    wcsncpy (t, s, n);
    // code ...
}
```

The following code corrects this warning by using the safe string manipulation function:

```
void f(wchar_t* t, wchar_t* s, size_t n)
{
    // code...
    wcsncpy_s( t, sizeof(s), s, n );
    // code...
}
```

## See Also

- [\\_strncpy\\_s, \\_strncpy\\_s\\_l, \\_wcsncpy\\_s, \\_wcsncpy\\_s\\_l, \\_mbsncpy\\_s, \\_mbsncpy\\_s\\_l](#)
- [sizeof Operator](#)

# C6384

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6384: dividing sizeof a pointer by another value

This warning indicates that a size calculation might be incorrect. To calculate the number of elements in an array, one sometimes divides the size of the array by the size of the first element; however, when the array is actually a pointer, the result is typically different than intended.

If the pointer is a function parameter and the size of the buffer was not passed, it is not possible to calculate the maximum buffer available. When the pointer is allocated locally, the size used in the allocation should be used.

## Example

The following code generates this warning:

```
#include <windows.h>
#include <TCHAR.h>

#define SIZE 15

void f( )
{
    LPTSTR dest = new TCHAR[SIZE];
    char src [SIZE] = "Hello, World!!";
    if (dest)
    {
        _tcsncpy(dest, src, sizeof dest / sizeof dest[0]);
    }
}
```

To correct this warning, pass the buffer size as shown in the following code:

```
#include <windows.h>
#include <TCHAR.h>

#define SIZE 15

void f( )
{
    LPTSTR dest = new TCHAR[SIZE];
    char src [SIZE] = "Hello, World!!";
    if (dest)
    {
        _tcsncpy(dest, src, SIZE);
    }
}
```

To correct this warning using the safe string function `_tcsncpy_s`, use the following code:

```
void f( )
{
    LPTSTR dest = new TCHAR[SIZE];
    char src [SIZE] = "Hello, World!!";
    if (dest)
    {
        _tcsncpy_s(dest, SIZE, src, SIZE);
    }
}
```

Note that the use of `new` and `delete` have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

## See Also

- [\\_mbsncpy\\_s, \\_mbsncpy\\_s\\_l](#)
- [sizeof Operator](#)

# C6385

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6385: invalid data: accessing <buffer name>, the readable size is <size1> bytes, but <size2> bytes may be read: Lines: x, y

This warning indicates that the readable extent of the specified buffer might be smaller than the index used to read from it. Attempts to read data outside the valid range leads to buffer overrun.

## Example

The following code generates this warning:

```
void f(int i)
{
    char a[20];
    char j;
    if (i <= 20)
    {
        j = a[i];
    }
}
```

To correct this warning, use the following code:

```
void f(int i)
{
    char a[20];
    char j;
    if (i < 20)
    {
        j = a[i];
    }
}
```

# C6386

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6386: buffer overrun: accessing <buffer name>, the writable size is <size1> bytes, but <size2> bytes may be written: Lines: x, y

This warning indicates that the writable extent of the specified buffer might be smaller than the index used to write to it. This can cause buffer overrun.

## Example

The following code generates both this warning and [C6201](#):

```
#define MAX 25

void f ( )
{
    char ar[MAX];
    //Code ...
    ar[MAX] = '\0';
}
```

To correct both warnings, use the following code:

```
#define MAX 25

void f ( )
{
    char a[MAX];
    // code...
    a[MAX - 1] = '\0';
}
```

## See Also

[C6201](#)

# C6387

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6387: <argument> may be <value>; this does not adhere to the specification for the function <function name>; Lines: x, y

This warning is raised if an annotated function parameter is being passed an unexpected value. For example, passing a potentially null value to a parameter that is marked with `_In_` annotation generates this warning.

## Example

The following code generates this warning because a null parameter is passed to `f(char *)`:

```
#include <sal.h>

_Post_ _Null_ char * g();

void f(_In_ char *pch);

void main()
{
    char *pCh = g();
    f(pCh); // Warning C6387
}
```

To correct this warning, use the following code:

```
#include <sal.h>

_Post_ _Notnull_ char * g();

void f(_In_ char *pch);

void main()
{
    char *pCh = g();
    f(pCh);
}
```

## See Also

[strlen](#), [wcslen](#), [\\_mbslen](#), [\\_mbslen\\_l](#), [\\_mbstrlen](#), [\\_mbstrlen\\_l](#)

# C6388

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6388: <argument> may not be <value>: this does not adhere to the specification for the function  
<function name>: Lines: x, y

This warning indicates that an unexpected value is being used in the specified context. This is typically reported for values passed as arguments to a function that does not expect it.

## Example

The following C++ code generates this warning because DoSomething expects a null value but a potentially non-null value might be passed:

```
#include <string.h>
#include <malloc.h>
#include <sal.h>

void DoSomething( _Pre_ _Null_ void* pReserved );

void f()
{
    void* p = malloc( 10 );
    DoSomething( p ); // Warning C6388
    // code...
    free(p);
}
```

To correct this warning, use the following sample code:

```
#include <string.h>
#include <malloc.h>
#include <sal.h>

void DoSomething( _Pre_ _Null_ void* pReserved );
void f()
{
    void* p = malloc( 10 );
    if (!p)
    {
        DoSomething( p );
    }
    else
    {
        // code...
        free(p);
    }
}
```

Note that the use of malloc and free have many pitfalls in terms of memory leaks and exceptions. To avoid these kinds of leaks and exception problems altogether, use the mechanisms that are provided by the C++ Standard Template Library (STL). These include [shared\\_ptr](#), [unique\\_ptr](#), and [vector](#). For more information, see [Smart Pointers](#) and [C++ Standard Library](#).

# C6400

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6400: Using <function name> to perform a case-insensitive compare to constant string <string name>. Yields unexpected results in non-English locales

This warning indicates that a case-insensitive comparison to a constant string is being performed in a locale-dependent way, when, apparently, a locale-independent comparison was intended.

The typical consequence of this defect is incorrect behavior in non-English speaking locales. For example, in Turkish, ".gif" will not match ".GIF"; in Vietnamese, "LogIn" will not match "LOGIN".

String comparisons should typically be performed with the `CompareString` function. To perform a locale-independent comparison on Windows XP, the first parameter should be the constant `LOCALE_INVARIANT`.

## Example

The following code generates this warning:

```
#include <windows.h>
int f(char *ext)
{
    // code...
    return (lstrcmpi(ext, TEXT("gif")) == 0);
}
```

To correct this warning, perform a locale-independent test for whether `char *ext` matches "gif" ignoring upper/lower case differences, use the following code:

```
#include <windows.h>
int f(char *ext)
{
    // code...
    return (CompareString(
        LOCALE_INVARIANT,
        NORM_IGNORECASE,
        ext,
        -1,
        TEXT ("gif"),
        -1) == CSTR_EQUAL);
}
```

## See Also

[CompareString](#)

# C6401

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6401: Using <function name> in a default locale to perform a case-insensitive compare to constant string < string name>. Yields unexpected results in non-English locales

This warning indicates that a case-insensitive comparison to a constant string is being performed when specifying the default locale; usually, a locale-independent comparison was intended.

The typical consequence of this defect is incorrect behavior in non-English speaking locales. For example, in Turkish, ".gif" will not match ".GIF"; in Vietnamese, "LogIn" will not match "LOGIN".

The `CompareString` function takes a locale as an argument; however, passing in a default locale, for example, the constant `LOCALE_USER_DEFAULT`, will cause different behaviors in different locales, depending on the user's default. Usually, case-insensitive comparisons against a constant string should be performed in a locale-independent comparison.

To perform a locale-independent comparison using `CompareString` on Windows XP, the first parameter should be the constant `LOCALE_INVARIANT`; for example, to perform a locale-independent test for whether `pString` matches `file1.gif` ignoring upper/lower case differences, use a call such as:

```
CompareString(LOCALE_INVARIANT,
              NORM_IGNORECASE,
              pString,
              -1,
              TEXT("file1.gif"),
              -1) == CSTR_EQUAL
```

## Example

The following code generates this warning:

```
include <windows.h>

int fd(char *ext)
{
    return (CompareString(LOCALE_USER_DEFAULT,
                          NORM_IGNORECASE,
                          ext,
                          -1,
                          TEXT("gif"),
                          -1) == 2);
}
```

To correct this warning, use the following code:

```
include <windows.h>
int f(char *ext)
{
    return (CompareString(LOCALE_INVARIANT,
        NORM_IGNORECASE,
        ext,
        -1,
        TEXT("gif"),
        -1) == 2);

}
```

## See Also

[CompareString](#)

# C6411

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning C6411: Potentially reading invalid data from the buffer.

This warning indicates that the value of the index that is used to read from the buffer can exceed the readable size of the buffer. Because the code analysis tool reports this warning when it cannot reduce a complex expression that represents the buffer size, or the index used to access the buffer, this warning might be reported in error.

## Example

The following code generates this warning.

```
char *a = new char[strlen(InputParam)];
delete[] a;
a[10];
```

The following code corrects this error.

```
int i = strlen(InputParam);
char *a = new char[i];
if (i > 10) a[10];
delete[] a;
```

# C6412

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6412: Potential buffer overrun while writing to buffer. The writable size is *write\_size* bytes, but *write\_index* bytes may be written.

This warning indicates that the value of the index that is used to write to the buffer can exceed the writeable size of the buffer.

Because the code analysis tool reports this warning when it cannot reduce a complex expression that represents the buffer size, or the index used to access the buffer, this warning might be reported in error.

## Example

The following code generates this warning.

```
char *a = new char[strlen(InputParam)];  
a[10] = 1;  
delete[] a;
```

The following code corrects this error.

```
int i = strlen(InputParam);  
char *a = new char[i];  
if (i > 10) a[10] = 1;  
delete[] a;
```

# C6500

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6500: invalid annotation: value for <name> property is invalid

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates that a property value used in the annotation is not valid. For example, it can occur if an incorrect level of dereference is used in the Deref property, or if you use a constant value that is larger than size\_t for properties like ElementSize.

## Example

The following code generates this warning because an incorrect level of dereference is used in the Pre condition:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pre( Deref=2, Access=SA_Read )] char buffer[] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;

void f( [Pre( Deref=2, Access=Read )] char buffer[] );
```

To correct this warning, specify the correct level of dereference, as shown in the following sample code:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pre( Deref=1, Access=SA_Read )] char buffer[] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;

void f( [Pre( Deref=1, Access=Read )] char buffer[] );
```

This warning is generated for both Pre and Post conditions.

# C6501

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6501: annotation conflict: <name> property conflicts with previously specified property

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates the presence of conflicting properties in the annotation. This typically occurs when multiple properties that serve similar purpose are used to annotate a parameter or return value. To correct the warning, you must choose the property that best addresses your need.

## Example

The following code generates this warning because both `ValidElementsConst` and `ValidBytesConst` provide a mechanism to allow valid data to be read:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void fd([SA_Pre(ValidElementsConst =4, ValidBytesConst =4)] char pch[]);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f( [Pre(ValidElementsConst=4, ValidBytesConst=4 )] char pch[ ] );
```

To correct this warning, use the most appropriate property, as shown in the following code:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pre(ValidElementsConst=4)] char pch[ ] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f( [Pre(ValidElementsConst=4)] char pch[ ] );
```

# C6503

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6503: Invalid annotation: references and arrays may not be marked Null=Yes or Null=Maybe

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates that Null property is incorrectly used on a reference or array type. A reference or array type holds the address of an object and must point to a valid object. Because reference and array types cannot be null, you must correct the error by either removing the Null property or by setting the Null property value to No.

## Example

The following code generates this warning:

```
// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
class Point
{
public:
    // members
};

void f([Pre(Null=Yes)] Point& pt);
```

To correct this warning, set the Null property to No as shown in the following code:

```
// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;

class Point
{
public:
    // members
};
void f([Pre(Null=No)] Point& pt);
```

# C6504

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6504: invalid annotation: property may only be used on values of pointer, pointer-to-member, or array type

This warning indicates the use of a property on an incompatible data type. For more information about data types supported by properties, see [Annotation Properties](#).

## Example

The following code generates this warning because the `_Null_` property cannot be used on the reference data type.

```
#include<sal.h>

class Point
{
public:
    // members
};

void f(_Pre_ _Null_ Point& pt)
{
    // code ...
}
```

To correct this warning, use the following code:

```
#include<sal.h>

class Point
{
public:
    // members
};

void f(_Pre_ _Null_ Point* pt)
{
    // code ...
}
```

The defective code shown earlier also generates warning [C6516](#) because property conflicts resulted in an invalid annotation.

# C6505

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6505: invalid annotation: MustCheck property may not be used on values of void type

This warning indicated that MustCheck property was used on a void data type. You cannot use MustCheck property on void type. Either remove the MustCheck property or use another data type.

## Example

The following code generates this warning:

```
#include <sal.h>
_Must_inspect_result_ void f()
{
    //Code ...
}
```

To correct this warning, use the following code:

```
#include <sal.h>
_Must_inspect_result_ char* f()
{
    char *str ="Hello World";
    //Code ...
    return str;
}
```

## See Also

[C6516](#)

# C6506

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6506: invalid annotation: <name> property may only be used on values of pointer or array types

This warning indicates that a property is used on a type other than pointer or array types. The Access, Tainted, and Valid properties can be used on all data types. Other properties, such as ValidBytesConst, ValidElementsConst, ElementSize, and NullTerminated support pointer, pointer to members, or array types. For a complete list of properties and the supported data types, see [Using SAL Annotations to reduce code defects](#).

## Example

The following code generates this warning:

```
#include<sal.h>
void f(_Out_ char c)
{
    c = 'd';
}
```

To correct this warning, use a pointer or an array type, as shown in the following sample code:

```
#include<sal.h>
void f(_Out_ char *c)
{
    *c = 'd';
}
```

## See Also

[C6516](#)

# C6508

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6508: invalid annotation: write access is not allowed on const values

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates that the Access property specified on a const parameter implies that it can be written to. For constant values, Access=Read is the only valid setting.

## Example

The following code generates this warning:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void fD ([SA_Pre(Deref=1,Access=SA_Write)]const char *pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(Deref=1,Access=Write)]const char *pc);
```

To correct this warning, use the following code:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(Deref=1,Access=SA_Read)]const char *pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(Deref=1,Access=Read)]const char *pc);
```

# C6509

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6509: invalid annotation: 'return' cannot be referenced from a precondition

This warning indicates that the `return` keyword cannot be used in a precondition. The `return` keyword is used to terminate the execution of a function and return control to the calling function.

## Example

The following code generates this warning because `return` is used in a precondition:

```
#include <sal.h>

int f (_In_reads_(return) char *pc)
{
    // code ...
    return 1;
}
```

To correct this warning, use the following code:

```
#include <sal.h>

int f (_In_reads_(i) char *pc, int i)
{
    // code ...
    return 1;
}
```

# C6510

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6510: Invalid annotation: 'NullTerminated' property may only be used on buffers whose elements are of integral or pointer type: Function '<function>' <parameter>.

This warning indicates an incorrect use of the **NullTerminated** property (those ending in '`_z`'). You can only use this type of property on pointer or array types.

## Example

The following code generates this warning:

```
#include <sal.h>

void f(_In_z_ char x)
{
    // code ...
}
```

To correct this warning, use the following code:

```
#include <sal.h>

void f(_In_z_ char * x)
{
    // code ...
}
```

## See Also

[C6516](#)

# C6511

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6511: invalid annotation: MustCheck property must be Yes or No

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates an invalid value for MustCheck property was specified. The only valid values for this property are: Yes and No.

## Example

The following code generates this warning:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
[returnvalue:SA_Post(MustCheck=SA_Maybe)] int f();

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
[returnvalue:Post(MustCheck=Maybe)] int f();
```

To correct this warning, a valid value for MustCheck property is used in the following code:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
[returnvalue:SA_Post(MustCheck=SA_Yes)] int f();

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
[returnvalue:Post(MustCheck=Yes)] int f();
```

## See Also

- [Using SAL Annotations to reduce code defects](#)
- [C6516](#)

# C6513

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6513: invalid annotation: ElementSizeConst requires additional size properties

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates that ElementSizeConst requires other properties that are missing from the annotation. Specifying ElementSizeConst alone does not provide any benefit to the analysis process. In addition to specifying ElementSize, other properties such as ValidElementsConst or WritableElementsConst must also be specified.

## Example

The following code generates this warning:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(ElementSizeConst=4)] void* pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(ElementSizeConst=4)] void* pc);
```

To correct this warning, use the following code:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(ElementSizeConst=4, ValidElementsConst=2)] void* pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(ElementSizeConst=4, ValidElementsConst=2)] void* pc);
```

Incorrect use of ElementSize property also generates this warning.

## See Also

[Using SAL Annotations to reduce code defects](#)

# C6514

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6514: invalid annotation: value of the <name> property exceeds the size of the array

This warning indicates that a property value exceeds the size of the array specified in the parameter being annotated. This warning occurs when the value specified for the annotation property is greater than the actual length of the array being passed.

## Example

The following code generates this warning because the size of the array is 6 whereas the ValidElementsConst property value is 8:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pref(Deref=1, ValidElementsConst=8)] char(*matrix) [6] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f( [Pref(Deref=1, ValidElementsConst=8)] char(*matrix) [6] );
```

To correct this warning, make sure the size of specified in ValidElementsConst is less than or equal to the size of the array, as shown in the following sample code:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f( [SA_Pref(Deref=1, ValidElementsConst=6)] char(*matirx) [6] );

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f( [Pref(Deref=1, ValidElementsConst=6)] char(*matirx) [6] );
```

# C6515

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6515 - invalid annotation: <name> property may only be used on values of pointer type

This warning indicates that a property for use on pointers was applied to a non-pointer type. For a list of annotation properties, see [Using SAL Annotations to reduce code defects](#).

## Example

The following code generates this warning:

```
#include <sal.h>

void f(_Readable_bytes_(c) char pc,  size_t c)
{
    // code ...
}
```

To correct this warning, use the following code:

```
#include <sal.h>

void f(_Readable_bytes_(c) char * pc,  size_t c)
{
    // code ...
}
```

## See Also

[C6516](#)

# C6516

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6516: invalid annotation: no properties specified for <name> attribute

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates that either no property was specified in the attribute or the property that was specified is invalid; therefore, the attribute cannot be considered complete.

## Example

The following code generates this warning because Deref=1 only specifies the level of indirection, but this information alone does not help the analysis tool:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pre(Deref=1)] char* pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(Deref=1)] char* pc);
```

To correct this warning, another property, such as Access, is required to indicate to the analysis tool what must be enforced on the de-referenced items. The following code corrects this warning:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pre(Deref=1, Access=SA_Read)] char* pc);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(Deref=1, Access=Read)] char* pc);
```

# C6517

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6517: Invalid annotation: 'SAL\_readableTo' property may not be specified on buffers that are not readable: '\_Param\_(1)'.

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates that `SAL_readableTo` property does not have the required read access. You cannot use this property to annotate a parameter without providing read access.

## Example

The following code generates this warning because read access is not granted on the buffer:

```
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre( ValidBytesConst=10 )][Pre( Deref=1, Access=Write )] char* buffer );
```

To correct this warning, grant read access as shown in the following code:

```
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre( ValidBytesConst=10 )][Pre( Deref=1, Access=Read)] char* buffer );
```

## See Also

[Using SAL Annotations to reduce code defects](#)

# C6518

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6518: Invalid annotation: 'SAL\_writableTo' property may not be specified as a precondition on buffers that are not writable: '\_Param\_(1)'

This warning indicates that a conflict exists between a `SAL_writableTo` property value and a writable property. This ordinarily indicates that a writable property does not have write access to the parameter being annotated.

## Example

The following code generates this warning because the `_out_` annotation compiles to include a `SAL_writableTo` property, which does not allow write access:

```
#include <sal.h>
void f(_Out_ const char* pc)
{
    //code that can't write to *pc ...
}
```

To correct this warning, use the following code:

```
#include <sal.h>
void f(_Out_ char* pc)
{
    pc = "Hello World";
    //code ...
}
```

# C6522

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6522: invalid size specification: expression must be of integral type

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates that an integral type was expected, but an incorrect data type was used. You can use annotation properties that accept the size of a parameter in terms of another parameter, but you must use correct data type. For a list of annotation properties, see [Using SAL Annotations to reduce code defects](#).

## Example

The following code generates this warning:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(ValidBytes="c")] char *pc, double c);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(ValidBytes="c")] char *pc, double c);
```

To correct this warning, use `size_t` for the `ValidBytesParam` parameter data type, as shown in the following sample code:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f ([SA_Pre(ValidBytes="c")] char *pc, size_t c);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f ([Pre(ValidBytes="c")] char *pc, size_t c);
```

# C6525

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6525: invalid size specification: property value may not be valid

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates that the property value used to specify the size is not valid. This occurs if the size parameter is annotated using Valid=No.

## Example

The following code generates this warning because the ValidElements property uses a size parameter that is marked not valid:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pre(ValidElements="*count")] char * px, [SA_Pre(Valid=SA_No)]size_t *count);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(ValidElements="*count")] char * px, [Pre(Valid=No)]size_t *count);
```

To correct this warning, specify a valid size parameter as shown in the following code:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pre(ValidElements="*count")] char * px, [SA_Pre(Valid=SA_Yes)]size_t *count);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(ValidElements="*count")] char * px, [Pre(Valid=Yes)]size_t *count);
```

## See Also

[Using SAL Annotations to reduce code defects](#)

# C6527

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6527: Invalid annotation: NeedsRelease property may not be used on values of void type

# C6530

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 6530: unrecognized format string style <name>

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

This warning indicates that the `FormatString` property is using a value other than `scanf` or `printf`. To correct this warning, review your code and use a valid value for the `Style` property.

## Example

The following code generates this warning because of a typo in the `Style` property:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_FormatString(Style="printfd")] char *px);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([FormatString(Style="printfd")] char *px);
```

To correct this warning, use a valid `Style` as shown in the following code:

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_FormatString(Style="printf")] char *px);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([FormatString(Style="printf")] char *px);
```

# C6540

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6540: The use of attribute annotations on this function will invalidate all of its existing `__declspec` annotations

# C6551

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6551: Invalid size specification: expression not parsable

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

# C6552

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6552: Invalid `Deref=` or `Notref=`: expression not parseable

## NOTE

This warning occurs only in code that is using a deprecated version of the source-code annotation language (SAL). We recommend that you port your code to use the latest version of SAL. For more information, see [Using SAL Annotations to Reduce C/C++ Code Defects](#).

# C6701

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6701: The value is not a valid Yes/No/Maybe value: <string>

This warning is reported when there is an error in the annotations.

# C6702

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6702: The value is not a string value: <string>

This warning is reported when there is an error in the annotations.

# C6703

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6703: The value is not a number: <string>

This warning is reported when there is an error in the annotations.

# C6704

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6704: Unexpected Annotation Expression Error: <annotation> [<why>]

This warning is reported when there is an error in the annotations.

# C6705

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6705: Annotation error expected <expected\_number> arguments for annotation <parameter> found <actual\_number>.

This warning is reported when there is an error in the annotations.

# C6706

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6706: Unexpected Annotation Error for annotation <annotation>: <why>

This warning is reported when there is an error in the annotations.

# C6707

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6707: Unexpected Model Error: <why>

# C6993

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 6993: Code analysis ignores OpenMP constructs; analyzing single-threaded code

This warning indicates that the Code Analyzer has encountered Open MP pragmas that it cannot analyze.

# C6995

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C6995: Failed to save XML Log file

This warning indicates that the Code Analysis tool cannot create the defect log, which is the output of the code analysis.

This error might indicate a disk error or indicate that you do not have permission to create a file in the specified directory.

# C6997

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 6997: Annotations at this location are meaningless and will be ignored.

Annotations cannot be applied to `extern "C" {...}`. Apply the annotations to a specific object.

## See Also

[Using SAL Annotations to Reduce C/C++ Code Defects](#)

# C26100

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26100: Race condition. Variable <var> should be protected by lock <lock>.

The `_Guarded_by_` annotation in the code specifies the lock to use to guard a shared variable. Warning C26100 is generated when the guard contract is violated.

## Example

The following example generates warning C26100 because there is a violation of the `_Guarded_by_` contract.

```
CRITICAL_SECTION gCS;

__Guarded_by__(gCS) int gData;

typedef struct _DATA {
    __Guarded_by__(cs) int data;
    CRITICAL_SECTION cs;
} DATA;

void Safe(DATA* p) {
    EnterCriticalSection(&p->cs);
    p->data = 1; // OK
    LeaveCriticalSection(&p->cs);
    EnterCriticalSection(&gCS);
    gData = 1; // OK
    LeaveCriticalSection(&gCS);
}

void Unsafe(DATA* p) {
    EnterCriticalSection(&p->cs);
    gData = 1; // Warning C26100 (wrong lock)
    LeaveCriticalSection(&p->cs);
}
```

The contract violation occurs because an incorrect lock is used in the function `Unsafe`. In this case, `gcs` is the correct lock to use.

## Example

Occasionally a shared variable only has to be guarded for write access but not for read access. In that case, use the `_Write_guarded_by_` annotation, as shown in the following example.

```
CRITICAL_SECTION gCS;

_Guarded_by_(gCS) int gData;

typedef struct _DATA2 {
    _Write_guarded_by_(cs) int data;
    CRITICAL_SECTION cs;
} DATA2;

int Safe2(DATA2* p) {
    // OK: read does not have to be guarded
    int result = p->data;
    return result;
}

void Unsafe2(DATA2* p) {
    EnterCriticalSection(&gCS);
    // Warning C26100 (write has to be guarded by p->cs)
    p->data = 1;
    LeaveCriticalSection(&gCS);
}
```

This example also generates warning C26100 because it uses an incorrect lock in the function `Unsafe2`.

# C26101

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26101: Failing to use interlocked operation properly for variable <var>.

Windows APIs offer a variety of interlocked operations. Annotation `_Interlocked_` specifies that a variable should only be accessed through an interlocked operation. Warning C26101 is issued when an access is not consistent with the `_Interlocked_` annotation.

## Example

The following example generates warning C26101 because there is a violation of the `_Interlocked_` contract.

```
CRITICAL_SECTION cs;
typedef struct _DATA
{
    _Interlocked_ LONG data;
} DATA;

void Safe(DATA* p)
{
    InterlockedIncrement(&p->data); // OK
}

void Unsafe(DATA* p)
{
    p->data += 1; // Warning C26101
    EnterCriticalSection(&cs);
    LeaveCriticalSection(&cs);
}
```

# C26105

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26105: Lock order violation. Acquiring lock <lock> with level <level> causes order inversion.

Concurrency SAL supports *lock levels*. To declare a lock level, which is denoted by a string literal without double quotes, use `_Create_lock_level_`. You can impose an order of acquisition between two lock levels by using the annotation `_Set_lock_level_order_(A,B)`, which states that locks that have level `A` must be acquired before locks that have level `B`. To establish a lock order hierarchy (a partial order among lock levels), use multiple `_Set_lock_level_order_` annotations. To associate a lock with a lock level, use the `_Set_lock_level_` annotation when you declare the lock. Warning C26105 is issued when a lock ordering violation is detected.

## Example

The following example generates warning C26105 because there is a lock order inversion in the function

`OrderInversion`.

```
_Create_lock_level_(MutexLockLevel);
_Create_lock_level_(TunnelLockLevel);
_Create_lock_level_(ChannelLockLevel);
_Lock_level_order_(MutexLockLevel, TunnelLockLevel);
_Lock_level_order_(TunnelLockLevel, ChannelLockLevel);
_Has_lock_level_(MutexLockLevel) HANDLE gMutex;

struct Tunnel
{
    _Has_lock_level_(TunnelLockLevel) CRITICAL_SECTION cs;
};

struct Channel
{
    _Has_lock_level_(ChannelLockLevel) CRITICAL_SECTION cs;
};

void OrderInversion(Channel* pChannel, Tunnel* pTunnel)
{
    EnterCriticalSection(&pChannel->cs);
    // Warning C26105
    WaitForSingleObject(gMutex, INFINITE);
    EnterCriticalSection(&pTunnel->cs);
    LeaveCriticalSection(&pTunnel->cs);
    LeaveCriticalSection(&pChannel->cs);
}
```

# C26110

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26110: Caller failing to hold lock <lock> before calling function <func>.

When a lock is required, make sure to clarify whether the function itself or its caller should acquire the lock.

Warning C26110 is issued when there is a violation of the `_Requires_lock_held_` annotation.

## Example

In the following example, warning C26110 is generated because the annotation `_Requires_lock_held_` on function `LockRequired` states that the caller of `LockRequired` must acquire the lock before it calls `LockRequired`. Without this annotation, `LockRequired` has to acquire the lock before it accesses any shared data protected by the lock.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    int d;
} DATA;

__Requires_lock_held_(p->cs)

void LockRequired(DATA* p)
{
    p->d = 0;
}

void LockNotHeld(DATA* p)
{
    LockRequired(p); // Warning C26110
}
```

# C26111

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26111: Caller failing to release lock <lock> before calling function <func>.

The annotation `_Requires_lock_not_held_` imposes a precondition that the lock count for the specified lock cannot be greater than zero when the function is called. Warning C26111 is issued when a function fails to release the lock before it calls another function.

## Example

The following example generates warning C26111 because the `_Requires_lock_not_held_` precondition is violated by the call to `DoNotLock` within the locked section.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    int d;
} DATA;

__Requires_lock_not_held_(p->cs)

void DoNotLock(DATA* p)
{
    EnterCriticalSection(&p->cs);
    p->d = 0;
    LeaveCriticalSection(&p->cs);
}

void LockedFunction(DATA* p)
{
    EnterCriticalSection(&p->cs);
    DoNotLock(p); // Warning C26111
    LeaveCriticalSection(&p->cs);
}
```

# C26112

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26112: Caller cannot hold any lock before calling <func>.

The annotation `_Requires_no_locks_held_` imposes a precondition that the caller must not hold any lock while it calls the function. Warning C26112 is issued when a function fails to release all locks before it calls another function.

## Example

The following example generates warning C26112 because the `_Requires_no_locks_held_` precondition is violated by the call to `NoLocksAllowed` within the locked section.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

__Requires_no_locks_held__

void NoLocksAllowed(DATA* p)
{
    // Lock sensitive operations here
}

void LocksHeldFunction(DATA* p)
{
    EnterCriticalSection(&p->cs);
    NoLocksAllowed(p); // Warning C26112
    LeaveCriticalSection(&p->cs);
}
```

# C26115

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26115: Failing to release lock <lock> in function <func>.

Enforcement of syntactically scoped lock *acquire* and lock *release* pairs in C/C++ programs is not performed by the language. A function may introduce a locking side effect by making an observable modification to the concurrency state. For example, a lock wrapper function increments the number of lock acquisitions, or lock count, for a given lock.

You can annotate a function that has a side effect from a lock acquire or lock release by using `_Acquires_lock_` or `_Releases_lock_`, respectively. Without such annotations, a function is expected not to change any lock count after it returns. If acquires and releases are not balanced, they are considered to be *orphaned*. Warning C26115 is issued when a function introduces an orphaned lock.

## Example

The following example generates warning C26115 because there is an orphaned lock in a function that is not annotated with `_Acquires_lock_`.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

void FailToReleaseLock(int flag, DATA* p)
{
    EnterCriticalSection(&p->cs);

    if (flag)
        return; // Warning C26115

    LeaveCriticalSection(&p->cs);
}
```

# C26116

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26116: Failing to acquire or to hold lock <lock> in <func>.

Enforcement of syntactically scoped lock *acquire* and lock *release* pairs in C/C++ programs is not performed by the language. A function may introduce a locking side effect by making an observable modification to the concurrency state. For example, a lock wrapper function increments the number of lock acquisitions, or lock count, for a given lock. You can annotate a function that has a side effect from a lock acquire or lock release by using `_Acquires_lock_` or `_Requires_lock_held`, respectively. Without such annotations, a function is expected not to change any lock count after it returns. If acquires and releases are not balanced, they are considered to be *orphaned*. Warning C26116 is issued when a function has been annotated with `_Acquires_lock_`, but it does not acquire a lock, or when a function is annotated with `_Requires_lock_held` and releases the lock.

## Example

The following example generates warning C26116 because the function `DoesNotLock` was annotated with `_Acquires_lock_` but does not acquire it. The function `DoesNotHoldLock` generates the warning because it is annotated with `_Requires_lock_held` and does not hold it.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

__Acquires_lock__(p->cs) void DoesLock(DATA* p)
{
    EnterCriticalSection(&p->cs); // OK
}

__Acquires_lock__(p->cs) void DoesNotLock(DATA* p)
{
    // Warning C26116
}

__Requires_lock_held__(p->cs) void DoesNotHoldLock(DATA* p)
{
    LeaveCriticalSection(&p->cs); // Warning C26116
}
```

## See Also

- [C26115](#)

# C26117

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26117: Releasing unheld lock <lock> in function <func>.

Enforcement of syntactically scoped lock *acquire* and lock *release* pairs in C/C++ programs is not performed by the language. A function may introduce a locking side effect by making an observable modification to the concurrency state. For example, a lock wrapper function increments the number of lock acquisitions, or lock count, for a given lock. You can annotate a function that has a side effect from a lock acquire or lock release by using

`_Acquires_lock_` or `_Releases_lock_`, respectively. Without such annotations, a function is expected not to change any lock count after it returns. If acquires and releases are not balanced, they are considered to be *orphaned*.

Warning C26117 is issued when a function that has not been annotated with `_Releases_lock_` releases a lock that it doesn't hold, because the function must own the lock before it releases it.

## Example

The following example generates warning C26117 because the function `ReleaseUnheldLock` releases a lock that it doesn't necessarily hold—the state of `flag` is ambiguous—and there is no annotation that specifies that it should.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

int flag;

void ReleaseUnheldLock(DATA* p)
{
    if (flag)
        EnterCriticalSection(&p->cs);
    // code ...
    LeaveCriticalSection(&p->cs);
}
```

## Example

The following code fixes the problem by guaranteeing that the released lock is also acquired under the same conditions.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

int flag;

void ReleaseUnheldLock(DATA* p)
{
    if (flag)
    {
        EnterCriticalSection(&p->cs);
        // code ...
        LeaveCriticalSection(&p->cs);
    }
}
```

## See Also

- [C26115](#)

# C26130

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26130: Missing annotation `_Requires_lock_held_(<lock>)` or `_No_competing_thread_at` function `<func>`. Otherwise it could be a race condition. Variable `<var>` should be protected by lock `<lock>`.

Warning C26130 is issued when the analyzer detects a potential race condition but infers that the function is likely to be run in a single threaded mode, for example, when the function is in the initialization stage based on certain heuristics.

## Example

In the following example, warning C26130 is generated because a `_Guarded_by_` member is being modified without a lock.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    _Guarded_by_(cs) int data;
} DATA;

void Init(DATA* p)
{
    p->data = 0; // Warning C26130
}
```

## Example

If the previous code is guaranteed to be operated in a single threaded mode, annotate the function by using `_No_competing_thread_`, as shown in the following example.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    _Guarded_by_(cs) int data;
} DATA;

_No_competing_thread_ void Init(DATA* p)
{
    p->data = 0; // Warning C26130 will be resolved
}
```

## Example

Alternatively, you can annotate a code fragment by using `_No_competing_thread_begin_` and `_No_competing_thread_end_`, as follows.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
    _Guarded_by_(cs) int data;
} DATA;

void Init(DATA* p)
{
    _No_competing_thread_begin_
    p->data = 0; // Warning C26130 will be resolved
    _No_competing_thread_end_
}
```

# C26135

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26135: Missing annotation <annotation> at function <func>.

Warning C26135 is issued when the analyzer infers that a function is a lock wrapper function that has a lock acquire or lock release side effect. If the code is not intended to be a wrapper function, then either the lock is leaking (if the lock is being acquired) or it is being released incorrectly (if the lock is being released).

## Example

The following example generates warning C26135 because an appropriate side effect annotation is missing.

```
typedef struct _DATA
{
    CRITICAL_SECTION cs;
} DATA;

void MyEnter(DATA* p)
{
    // Warning C26135:
    // Missing side effect annotation _Acquires_lock_(&p->cs)
    EnterCriticalSection(&p->cs);
}

void MyLeave(DATA* p)
{
    // warning C26135:
    // Missing side effect annotation _Releases_lock_(&p->cs)
    LeaveCriticalSection(&p->cs);
}
```

## Example

Warning C26135 is also issued when a conditional locking side effect is detected. To annotate a conditional effect, use the `_When_(ConditionExpr, LockAnnotation)` annotation, where `LockAnnotation` is either `_Acquires_lock_` or `_Releases_lock_` and the predicate expression `ConditionExpr` is a Boolean conditional expression. The side effects of other annotations on the same function only occur when `ConditionExpr` evaluates to true. Because `ConditionExpr` is used to relay the condition back to the caller, it must involve variables that are recognized in the calling context. These include function parameters, global or class member variables, or the return value. To see the return value, use a special keyword in the annotation, `return`, as shown in the following example.

```

typedef struct _DATA
{
    CRITICAL_SECTION cs;
    int state;
} DATA;

_When_(return != 0, _Acquires_lock_(p->cs))
int TryEnter(DATA* p)
{
    if (p->state != 0)
    {
        EnterCriticalSection(&p->cs);
        return p->state;
    }

    return 0;
}

```

For shared/exclusive locks, also known as reader/writer locks, you can express locking side effects by using the following annotations:

- `_Acquires_shared_lock_(LockExpr)`
- `_Releases_shared_lock_(LockExpr)`
- `_Acquires_exclusive_lock_(LockExpr)`
- `_Releases_exclusive_lock_(LockExpr)`

# C26138

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26138: Suspending a coroutine while holding lock <lock>.

Warning C26138 warns when a coroutine is suspended while holding a lock. In general, we can't know how long will a coroutine remain in the suspended state so this pattern may result in longer critical sections than expected.

## Example

The following code will generate C26138.

```
#include <experimental/generator>
#include <future>
#include <mutex>

using namespace std::experimental;

std::mutex global_m;
_Guarded_by_(global_m) int var = 0;

generator<int> mutex_acquiring_generator() {
    global_m.lock();
    ++var;
    co_yield 1;                                // @expected(26138), global_m is hold while yielding.
    global_m.unlock();
}

generator<int> mutex_acquiring_generator_report_once() {
    global_m.lock();
    ++var;
    co_yield 1;                                // @expected(26138), global_m is hold while yielding.
    co_yield 1;                                // @expected(26138), global_m is hold while yielding.
    global_m.unlock();
}
```

## Example

The following code will correct these warnings.

```
#include <experimental/generator>
#include <future>
#include <mutex>

using namespace std::experimental;

std::mutex global_m;
_Guarded_by_(global_m) int var = 0;

generator<int> mutex_acquiring_generator2() {
{
    global_m.lock();
    ++var;
    global_m.unlock();
}
co_yield 1;                                // no 26138, global_m is already released above.
}
```

# C26140

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26140: Undefined lock kind <lock> in annotation <annotation> on lock <lock>.

## Example

```
_Has_lock_kind_(MUTEXa) HANDLE gMutex;

struct CorrectExample
{
    _Has_lock_kind_(Lock_kind_mutex_) HANDLE mMutex;
    _Guarded_by_(mMutex) int mData;
};

_WHEN_(return == WAIT_OBJECT_0 || return == WAIT_ABANDONED, _Acquires_lock_(gMutex))
DWORD UndefinedLockKind() // Warning C26140
{
    DWORD result = WaitForSingleObject(gMutex, 1000);
    return result;
}
```

# C26160

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26160: Caller possibly failing to hold lock <lock> before calling function <func>.

Warning C26160 resembles warning [C26110](#) except that the confidence level is lower. For example, the function may contain annotation errors.

## Example

The following code generates warning C26160.

```
struct Account
{
    _Guarded_by_(cs) int balance;
    CRITICAL_SECTION cs;

    _No_competing_thread_ void Init()
    {
        balance = 0; // OK
    }

    _Requires_lock_held_(this->cs) void FuncNeedsLock();

    _No_competing_thread_ void FuncInitCallOk()
        // this annotation requires this function is called
        // single-threaded, therefore we don't need to worry
        // about the lock
    {
        FuncNeedsLock(); // OK, single threaded
    }

    void FuncInitCallBad() // No annotation provided, analyzer generates warning
    {
        FuncNeedsLock(); // Warning C26160
    }
};
```

## Example

The following code shows a solution to the previous example.

```
struct Account
{
    _Guarded_by_(cs) int balance;
    CRITICAL_SECTION cs;

    _No_competing_thread_ void Init()
    {
        balance = 0; // OK
    }

    _Requires_lock_held_(this->cs) void FuncNeedsLock();

    _No_competing_thread_ void FuncInitCallOk()
        // this annotation requires this function is called
        // single-threaded, therefore we don't need to worry
        // about the lock
    {
        FuncNeedsLock(); // OK, single threaded
    }

    void FuncInitCallBadFixed() // this function now properly acquires (and releases) the lock
    {
        EnterCriticalSection(&this->cs);FuncNeedsLock();LeaveCriticalSection(&this->cs);
    }
};
```

# C26165

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26165: Possibly failing to release lock <lock> in function <func>.

Warning C26165 resembles warning [C26115](#) except that the confidence level is lower. For example, the function may contain annotation errors.

## Example

The following code generates warning C26165.

```
_Create_lock_level_(LockLevelOne);
_Create_lock_level_(LockLevelTwo);

struct LockLevelledStruct
{
    _Has_lock_level_(LockLevelOne) CRITICAL_SECTION a;
    _Has_lock_level_(LockLevelTwo) CRITICAL_SECTION b;
};

_Lock_level_order_(LockLevelOne, LockLevelTwo);

_Acquires_lock_(s->b) void GetLockFunc(LockLevelledStruct* s)
{
    EnterCriticalSection(&s->b);
}

void testLockLevelledStruct(LockLevelledStruct* s) // Warning C26165
{
    EnterCriticalSection(&s->a);
    GetLockFunc(s);
    LeaveCriticalSection(&s->a);
}
```

## Example

To correct this warning, change the previous example to the following.

```
_Create_lock_level_(LockLevelOne);
_Create_lock_level_(LockLevelTwo);

struct LockLevelledStruct
{
    _Has_lock_level_(LockLevelOne) CRITICAL_SECTION a;
    _Has_lock_level_(LockLevelTwo) CRITICAL_SECTION b;
};

_Lock_level_order_(LockLevelOne, LockLevelTwo);

_Acquires_lock_(s->b) void GetLockFunc(LockLevelledStruct* s)
{
    EnterCriticalSection(&s->b);
}

_Releases_lock_(s->b) void ReleaseLockFunc(LockLevelledStruct* s)
{
    LeaveCriticalSection(&s->b);
}

void testLockLevelledStruct(LockLevelledStruct* s) // OK
{
    EnterCriticalSection(&s->a);
    GetLockFunc(s);
    ReleaseLockFunc(s);
    LeaveCriticalSection(&s->a);
}
```

# C26166

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26166: Possibly failing to acquire or to hold lock <lock> in function <func>.

Warning C26166 resembles warning [C26116](#) except that the confidence level is lower. For example, the function may contain annotation errors.

## Example

The following code shows code that will generate warning C26166.

```
typedef struct _DATA {
CRITICAL_SECTION cs;
} DATA;

_Acquires_lock_(p->cs) void Enter(DATA* p) {
    EnterCriticalSection(&p->cs); // OK
}

_Acquires_lock_(p->cs) void BAD(DATA* p) {} // Warning C26166
```

# C26167

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26167: Possibly releasing unheld lock <lock> in function <func>.

Warning C26167 resembles warning [C26117](#) except that the confidence level is lower. For example, the function may contain annotation errors.

## Example

The following code will generate C26167, as well as C26110.

```
typedef struct _DATA {
    CRITICAL_SECTION cs;
} DATA;

_Releases_lock_(p->cs) void Leave(DATA* p) {
    LeaveCriticalSection(&p->cs); // OK
}
void ReleaseUnheldLock(DATA* p) { // Warning C26167
    int i = 0;
    Leave(p); // Warning C26110
}
```

## Example

The following code will correct these warnings.

```
typedef struct _DATA {
    CRITICAL_SECTION cs;
} DATA;

_Releases_lock_(p->cs) void Leave(DATA* p) {
    LeaveCriticalSection( &p->cs );
}

void ReleaseUnheldLock( DATA* p ) {
    EnterCriticalSection( &p->cs );
    int i = 0;
    Leave(p);
}
```

# C26800

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26800: Use of a moved from object: <lock>.

Warning C26800 is triggered when variable is used after it has been moved from. A variable is considered moved from after it was passed to a function as rvalue reference. There are some legitimate exceptions for uses such as assignment, destruction, and some state resetting functions such as std::vector::clear.

## Example

The following code will generate C26800.

```
#include <utility>

struct X {
    X();
    X(const X&);
    X(X&&);
    X &operator=(X&);
    X &operator=(X&&);
    ~X();
};

template<typename T>
void use_cref(const T&);

void test() {
    X x1;
    X x2 = std::move(x1);
    use_cref(x1);           // @expected(26800)
}
```

## Example

The following code will not generate C26800.

```
#include <utility>

struct MoveOnly {
    MoveOnly();
    MoveOnly(MoveOnly&) = delete;
    MoveOnly(MoveOnly&&);
    MoveOnly &operator=(MoveOnly&) = delete;
    MoveOnly &operator=(MoveOnly&&);
    ~MoveOnly();
};

template<typename T>
void use(T);

void test() {
    MoveOnly x;
    use(std::move(x)); // no 26800
}
```

# C26810

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26810: Lifetime of captured variable <var> might end by the time the coroutine is resumed.

Warning C26810 is triggered when a memory region might be used after it went out of scope in a resumed coroutine.

## Example

The following code will generate C26810.

```
#include <experimental/generator>
#include <future>

using namespace std::experimental;

coroutine_handle<> g_suspended_coro;

// Simple awaiter to allows to resume a suspended coroutine
struct ManualControl
{
    coroutine_handle<>& save_here;

    bool await_ready() { return false; }
    void await_suspend(coroutine_handle<> h) { save_here = h; }
    void await_resume() {}
};

void bad_lambda_example1()
{
    int x = 5;
    auto bad = [x]() -> std::future<void> {
        co_await ManualControl{g_suspended_coro}; // @expected(26810), Lifetime of capture 'x' might end by the
                                                // time this coroutine is resumed.
        printf("%d\n", x);
    };
    bad();
}
```

## See Also

- [C26811](#)

# C26811

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C26811: Lifetime of the memory referenced by parameter <var> might end by the time the coroutine is resumed.

Warning C26811 is triggered when a memory region might be used after it went out of scope in a resumed coroutine.

## Example

The following code will generate C26811.

```
#include <experimental/generator>
#include <future>

using namespace std::experimental;

// Simple awaiter to allows to resume a suspended coroutine
struct ManualControl
{
    coroutine_handle<>& save_here;

    bool await_ready() { return false; }
    void await_suspend(coroutine_handle<> h) { save_here = h; }
    void await_resume() {}
};

coroutine_handle<> g_suspended_coro;

std::future<void> async_coro(int &a)
{
    co_await ManualControl{g_suspended_coro}; // @expected(26811), Lifetime of 'a' might end by the time this
                                                // coroutine is resumed.
    ++a;
}
```

## See Also

- [C26810](#)

# C28020

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28020: The expression <expr> is not true at this call

This warning is reported when the `_Satisfies_` expression listed is not true. Frequently this indicates an incorrect parameter.

If this occurs on a function declaration, the annotations indicate an impossible condition.

# C28021

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28021: The parameter <param> being annotated with <anno> must be a pointer

This warning is reported when the object being annotated is not a pointer type. This annotation cannot be used with `void` or integral types.

# C28022

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28022: The function class(es) <classlist1> on this function do not match the function class(es) <classlist2> on the typedef used to define it.

This warning is reported when there is an error in the annotations. Both the typedef and the function itself have `_Function_class_` annotations, but they do not match. If both are used they must match.

# C28023

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28023: The function being assigned or passed should have a `_Function_class_` annotation for at least one of the class(es) in: <classlist>

This warning is usually reported when only one function class is in use and a callback of the appropriate type is not declared.

This warning is issued when the function on the left side of the assignment (or of the implied assignment, if this is a function call) is annotated to indicate that it is a driver-specific function type that uses the `_Function_class_` annotation or a typedef that contains such an annotation. The function on the right side of the assignment does not have a `_Function_class_` annotation. The function on the right should be annotated to be of the same type as the function on the left. This is usually best done by adding the declaration <class1> <funcname1> before the current first declaration of <funcname2>.

# C28024

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28024: The function pointer being assigned to is annotated with the function class <class>, which is not contained in the function class(es) <classlist>.

This warning is reported when both functions were annotated with a function class, but the classes do not match.

this warning is issued when a function pointer has a `_Function_class_` annotation that specifies that only functions of a particular functional class should be assigned to it. In an assignment or implied assignment in a function call, the source and target must be of the same function class, but the function classes do not match.

# C28039

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28039: The type of actual parameter <operand> should exactly match the type <typename>

This warning is usually reported when an enum formal was not passed a member of the enum, but may also be used for other types.

Because C permits enums to be used interchangeably, and interchangeably with constants, it is easy to pass the wrong enum value to a function without an error.

For enum types, if the type of an enum parameter is annotated with `_Enum_is_bitflag_`, arithmetic is permitted on the parameter. Otherwise the parameter must be of exactly the correct type. If a constant is strictly required, warning C28137 may also apply.

This rule can be used for other parameter types as well; see the function documentation for why the types must match exactly.

# C28103

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28103: Leaking resource

The specified object contains a resource that has not been freed. A function being called has been annotated with `__drv_acquiresResource` or `__drv_acquiresResourceGlobal` and this warning indicates that the resource named in the annotation was not freed.

## Example

The following code example generates this warning:

```
res = KeSaveFloatingPointState(buffer);
```

The following code example avoids this warning:

```
res = KeSaveFloatingPointState(buffer);
if (NT_SUCCESS(res))
{
    res = KeRestoreFloatingPointState(buffer);
}
```

If this warning is reported as a false positive, the most likely cause is that the function that releases the resource is not annotated with `__drv_releasesResource` or `__drv_releasesResourceGlobal`. Note that if you are using wrapper functions for system functions, the wrapper functions should use the same annotations that the system functions do. Currently, many system functions are annotated in the model file, so the annotations are not visible in the header files.

# C28104

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28104: Resource that should have been acquired before function exit was not acquired

A function that is intended to acquire a resource before it exits has exited without acquiring the resource. This warning indicates that the function is annotated with `__drv_acquiresResource` but does not return having actually acquired the resource. If this function is a wrapper function, a path through the function did not reach the wrapped function. If the failure to reach the wrapped function is because the function returned an error and did not actually acquire the resource, you might need to use a conditional annotation (`__drv_when`).

If this function actually implements the acquisition of the resource, it might not be possible for PFD to detect that the resource is acquired. In that case, use a `#pragma` warning to suppress the error. You can probably place the `#pragma` on the line preceding the `{` that begins the function body. The calling functions still need the annotation, but the Code Analysis tool will not be able to detect that the resource was acquired.

## Example

```
__drv_acquireResourceGlobal(HWLock, lockid)
void GetHardwareLock(lockid)
#pragma warning (suppress: 28104)
{
    // code to implement a hardware lock (which the Code Analysis tool can't recognize)
}
```

# C28105

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning C28105: Leaking resource due to an exception

The specified resource is not freed when an exception is raised. The statement specified by the path can raise an exception. This warning is similar to warning [C28103](#), except that in this case an exception is involved.

## Example

The following code example generates this warning:

```
res = KeSaveFloatingPointState(buffer);

res = AllocateResource(Resource);
char *p2 = new char[10]; // could throw

delete[] p2;
FreeResource(Resource)
```

The following code example avoids this warning:

```
res = AllocateResource(Resource);
char *p2;

try {
    p2 = new char[10];
} catch (std::bad_alloc *e) {
    // just handle the throw
    ;
}
FreeResource(Resource)
```

# C28106

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning C28106: Variable already holds resource possibly causing leak

A variable that contains a resource is used in a context in which a new value can be placed in the variable. If this occurs, the resource can be lost and not properly freed, causing a resource leak.

## Example

The following code example generates this warning:

```
ExAcquireResourceLite(resource, true);
...
ExAcquireResourceLite(resource, true);
```

The following code example avoids this warning:

```
ExAcquireResourceLite(resource1, true);
...
ExAcquireResourceLite(resource2, true);
```

# C28107

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28107: Resource must be held when calling function

A resource that the program must acquire before calling the function was not acquired when the function was called. As a result, the function call will fail. This warning is reported only when resources are acquired and released in the same function.

## Example

The following code example generates this warning:

```
ExAcquireResourceLite(resource, true);
ExReleaseResourceLite(resource);
```

The following code example avoids this warning:

```
KeEnterCriticalSection();
ExAcquireResourceLite(resource, true);
ExReleaseResourceLite(resource);
KeLeaveCriticalSection();
KeEnterCriticalSection();
ExAcquireResourceLite(resource, true);
ExReleaseResourceLite(resource);
KeLeaveCriticalSection();
```

# C28108

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28108: Variable holds an unexpected resource

The resource that the driver is using is in the expected C language type, but has a different semantic type.

## Example

The following code example generates this warning:

```
KeAcquireInStackSpinLock(spinLock, lockHandle);
...
KeReleaseSpinLock(spinLock, 0);
```

The following code example avoids this warning:

```
KeAcquireInStackSpinLock(spinLock, lockHandle);
...
KeReleaseInStackSpinLock(lockHandle);
```

# C28109

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28109: Variable cannot be held at the time function is called

The program is holding a resource that should not be held when it is calling this function. Typically, it indicates that the resource was unintentionally acquired twice. The Code Analysis tool reports this warning when resources are acquired and released in the same function.

## Example

The following code example generates this warning:

```
ExAcquireResourceLite(resource, true);
...
ExAcquireResourceLite(resource, true);
```

The following code example avoids this warning:

```
ExAcquireResourceLite(resource, true);
```

# C28112

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28112: A variable which is accessed via an Interlocked function must always be accessed via an Interlocked function

See line [number]: It is not always safe to access a variable which is accessed via the Interlocked\* family of functions in any other way.

A variable that is accessed by using the Interlocked executive support routines, such as InterlockedCompareExchangeAcquire, is later accessed by using a different function. Although certain ordinary assignments, accesses, and comparisons to variables that are used by the Interlocked\* routines can be safely accessed by using a different function, the risk is great enough to justify examining each instance.

## Example

The following code example generates this warning:

```
inter_var --;  
...  
InterlockedIncrement(&inter_var);
```

The following code example avoids this warning:

```
InterlockedDecrement(&inter_var);  
...  
InterlockedIncrement(&inter_var);
```

# C28113

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28113: Accessing a local variable via an Interlocked function

The driver is using an Interlocked executive support routine, such as [InterlockedDecrement](#), to access a local variable.

Although drivers are permitted to pass the address of a local variable to another function, and then use an interlocked function to operate on that variable, it's important to verify that the stack will not be swapped out to disk unexpectedly and that the variable has the correct life time across all threads that might use it.

## Example

Typically, the return value of an Interlocked executive support routine is used in subsequent computations, instead of the input arguments. Also, the Interlocked routines only protect the first (leftmost) argument. Using an Interlocked routine in the following way does not protect the value of global and often serves no purpose.

```
InterlockedExchange(&local, global)
```

The following form has the same effect on the data and safely accesses the global variable.

```
local = InterlockedExchange(&global, global)
```

# C28125

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28125: The function must be called from within a try/except block

The driver is calling a function that must be called from within a try/except block, such as [ProbeForRead](#), [ProbeForWrite](#), [MmProbeAndLockPages](#).

## Example

The following code example generates this warning:

```
ProbeForRead(addr, len, 4);
```

The following code example avoids this warning:

```
_try
{
    ProbeForRead(addr, len, 4);
}
_except(EXCEPTION_EXECUTE_HANDLER)
{
    Status = GetExceptionCode();
    ... report error status
}
```

# C28137

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28137: The variable argument should instead be a (literal) constant

This warning is reported when a function call is missing a required (literal) constant. Consult the documentation for the function.

## Example

For example, the `ExAcquireResourceExclusiveLite` routine requires a value of TRUE or FALSE for the `Wait` parameter. The following example code generates this warning:

```
ExAcquireResourceExclusiveLite(Resource, Wait);
```

The following code example avoids this warning:

```
ExAcquireResourceExclusiveLite(Resource, TRUE);
```

# C28138

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28138: The constant argument should instead be variable

This warning is reported in a function call that expects a variable or a non-constant expression, but the call includes a constant. For information about the function and its parameter, consult the WDK documentation of the function.

## Example

For example, in the following code example, the parameter of the `READ_PORT_UCHAR` macro must be a pointer to the port address, not the address provided as a constant.

The following code example generates this warning message:

```
READ_PORT_UCHAR(0x80001234);
```

To correct this warning, use a pointer to the port address.

```
READ_PORT_UCHAR(PortAddress);
```

There are a few older devices for which a constant parameter is acceptable with the `READ_PORT` and `WRITE_PORT` family of functions. When those devices receive this warning, the warning can be suppressed or ignored. However, any new devices should not assume a constant hardware address.

# C28159

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28159: Consider using another function instead.

This warning is reported for Drivers is suggesting that you use a preferred function call that is semantically equivalent to the function that the driver is calling. This is a general warning message; the annotation `__drv_preferredFunction` was used (possibly with a conditional a `__drv_when` () annotation) to flag a bad coding practice.

## Example

The following code example generates this warning:

```
char buff[MAX_PATH];  
  
OemToChar(buff, input);  
  
    // if strlen(input) > MAX_PATH  
    ..... leads to buffer overrun
```

The following code example avoids this warning:

```
char buff[MAX_PATH];  
  
OemToCharBuff(buff, input, MAX_PATH);
```

# C28160

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28160: Error annotation

This warning is reported when a `__drv_error` annotation has been encountered. This annotation is used to flag coding practices that should be fixed, and can be used with a `__drv_when` annotation to indicate specific combinations of parameters.

# C28163

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28163: The function should never be called from within a try/except block

This warning is reported when a function is of a type that should never be enclosed in a `try/except` block is found in a `try/except` block. The code analysis tool found at least one path in which the function called was within a `try/except` block.

# C28164

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28164: The argument is being passed to a function that expects a pointer to an object (not a pointer to a pointer)

This warning is reported when a pointer to a pointer is used in a call to a function that is expecting a pointer to an object.

The function takes a PVOID in this position. Usually, this indicates that &pXXX was used when pXXX is required.

Some *polymorphic functions* (functions that can evaluate to, and be applied to, values of different types) are implemented in C by using a PVOID argument that takes any pointer type. However, this allows the programmer to code a pointer to a pointer without causing a compiler error, even when this type is not appropriate.

## Example

The following code example generates this warning:

```
PFAST_MUTEX pFm;  
...  
KeWaitForSingleObject(&pFm, UserRequest, UserMode, false, NULL);
```

The following code example avoids the warning:

```
PFAST_MUTEX pFm;  
...  
KeWaitForSingleObject(pFm, UserRequest, UserMode, false, NULL);
```

# C28182

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28182: Dereferencing NULL pointer.

**Additional information:** <pointer1> contains the same NULL value as <pointer2> did <note>

The code analysis tool reports this warning when it confirms that the pointer can be NULL. If there are unconfirmed instances where the error might occur earlier in the trace, the code analysis tool adds the line number of the first instance to the warning message so that you can change the code to address all instances.

<pointer2> is confirmed to be potentially NULL. <pointer1> contains the same value as pointer2 and is being dereferenced. Because these pointers may be at very different places in the code, both are reported so that you can determine why the code analysis tool is reporting this warning.

If an unconfirmed earlier instance of the condition exists, then <note> is replaced by this text: "See line <number> for an earlier location where this can occur."

## Example

The following example shows code that could cause the code analysis tool to generate this warning message. In this example, the code analysis tool determines that `pNodeFree` is NULL in the `if` statement, and the code path into the body of the `if` is taken. However, because `nBlockSize` is potentially zero, the body of the `for` statement is not executed and `pNodeFree` is left unmodified. `pNodeFree` is then assigned to `pNode`, and `pNode` is used while a NULL dereference could occur.

```
typedef struct xlist {
    struct xlist *pNext;
    struct xlist *pPrev;
} list;

list *pNodeFree;
list *masterList;
int nBlockSize;

void fun()
{
    if (pNodeFree == 0)
    {
        list *pNode = masterList;

        for (int i = nBlockSize-1; i >= 0; i--, pNode--)
        {
            pNode->pNext = pNodeFree;
            pNodeFree = pNode;
        }
    }

    list* pNode = pNodeFree;
    pNode->pPrev = 0;
}
```

The code analysis tool reports the following warning:

```
:\\sample\\testfile.cpp(24) : warning C28182: Dereferencing NULL pointer. 'pNode' contains the same NULL value  
as 'pNodeFree' did.: Lines: 12, 14, 16, 23, 24
```

## Example

One way to correct the earlier example is to check `pNode` for zero before dereferencing it so that a NULL dereference is averted. The following code shows this correction.

```
typedef struct xlist {  
    struct xlist *pNext;  
    struct xlist *pPrev;  
} list;  
  
list *pNodeFree;  
list *masterList;  
int nBlockSize;  
  
void fun()  
{  
    if (pNodeFree == 0)  
    {  
        list *pNode = masterList;  
  
        for (int i = nBlockSize-1; i >= 0; i--, pNode--)  
        {  
            pNode->pNext = pNodeFree;  
            pNodeFree = pNode;  
        }  
    }  
  
    list* pNode = pNodeFree;  
    if(pNode != 0)  
        pNode->pPrev = 0;  
}
```

# C28183

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28183: The argument could be one value, and is a copy of the value found in the pointer

This warning indicates that this value is unexpected in the current context. This warning usually appears when a `NULL` value is passed as an argument to a function that does not permit it. The value was actually found in the specified variable, and the argument is a copy of that variable.

The Code Analysis tool reports this warning at the first point where it can definitively determine that the pointer is `NULL` or that it contains an illegal value. However, it is often the case that the error could actually occur earlier in the trace. When this happens, the Code Analysis tool will also give the line number of the first possible instance -- usually at a location where it could not definitively determine that the warning was appropriate. In those cases, the earlier location where this can occur is appended to the warning message. Typically, a code change should occur at or before that line number, rather than at the point of report.

## Example

In the following example, the Code Analysis tool determines that `s` is `NULL` in the `if` statement, and the body of the `if` is taken. The pointer `s` is then assigned to `t` and then `t` is used in a way where a `NULL` dereference could occur.

```
#include <windows.h>

int fun2(char *s)
{
    char *t;
    if (s == NULL) {
        //... but s is unchanged
    }

    t = s;

    return lstrlenA(t);
}
```

For this code example, the Code Analysis tool reports the following warning:

**d:\sample\testfile.cpp(38) : warning C28183: 't' could be '0', and is a copy of the value found in 's': this does not adhere to the specification for the function 'lstrlenA'.: Lines: 31, 32, 36, 38**

# C28193

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28193: The variable holds a value that must be examined

This warning indicates that the calling function is not checking the value of the specified variable, which was supplied by a function. The returned value is annotated with the `_check_return_` annotation, but the calling function is either not using the value or is overwriting the value without examining it.

This warning is similar to warning [C6031](#), but it is reported only when the code does not test or examine the value of the variable, such as by using it in a comparison. Simply assigning the value is not considered to be a sufficient examination to avoid this warning. Aliasing the result out of the function is considered a sufficient examination, but the result itself should be annotated with `_Check_return_`.

Certain functions (such as `strlen`) exist almost exclusively for their return value, so it makes sense for them to have the `_Check_return_` annotation. For these functions, the Code Analysis tool might report this warning when the return value is unused. This usually indicates that the code is incorrect, for example, it might contain residual code that could be deleted. However, in some rare instances, the return value is intentionally not used. The most common of these instances is where a string length is returned but not actually used before some other test is made. That other test causes a path to be simulated where the string length ends up being unused. When this happens, the code can be correct, but it might be inefficient.

There are two primary strategies for dealing with these cases where the return value is unused:

Reorder the code so that the string length is only returned along the path where it is needed.

Use a `#pragma` warning to suppress the warning--if by reordering the code, you would make the code too complex or otherwise less useful.

## Example

The following code example generates this warning:

```
IoGetDmaAdapter(pPDO, &DevDesc, &nMapRegs);
...
```

The following code example avoids this warning:

```
IoGetDmaAdapter(pPDO, &DevDesc, &nMapRegs);
...
if (nMapRegs < MIN_REQUIRED_MAPS) {
...
}
```

# C28194

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28194: The function was declared as aliasing the value in variable and exited without doing so

This warning indicates that the function prototype for the function being analyzed has a `__drv_isAliased` annotation, which indicates that it will *alias* the specified argument (that is, assign the value in a way that it will survive returning from the function). However, the function does not alias the argument along the path that is indicated by the annotation. Most functions that alias a variable save its value to a global data structure.

# C28195

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28195: The function was declared as acquiring memory in a variable and exited without doing so

This warning indicates that the function prototype for the function being analyzed has a `__drv_acquiresMemory` annotation. The `__drv_acquiresMemory` annotation indicates that the function acquires memory in the designated result location, but in at least one path, the function did not acquire the memory. Note that the Code Analysis tool will not recognize the actual implementation of a memory allocator (involving address arithmetic) and will not recognize that memory is allocated (although many wrappers will be recognized). In this case, the Code Analysis tool does not recognize that the memory was allocated and issues this warning. To suppress the false positive, use a `#pragma` warning on the line that precedes the opening brace `{` of the function body

# C28196

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28196: The requirement is not satisfied. (The expression does not evaluate to true.)

This warning indicates that the function prototype for the function being analyzed has a `_notnull`, `_null` or `_drv_valueIs` on an `_out_` parameter or the return value, but the value returned is inconsistent with that annotation.

# C28197

2/8/2019 • 2 minutes to read • [Edit Online](#)

## Warning C28197: Possibly leaking memory

This warning is reported for both memory and resource leaks when the resource is potentially aliased to another location.

The *pointer* points to allocated memory or to another allocated resource that was not explicitly freed. This warning is usually due to inadequate annotations on the called function, although inadequate annotations on the calling function can also make this more likely.

This warning can be reported on function exit if an input argument has a `__drv_freesMem` or `__drv_aliasesMem` annotation. This warning typically indicates either a valid leak or that a function called by the current function needs additional annotation.

In particular, the absence of the basic `_In_` and `_Out_` annotations make this warning fairly likely, although the `__drv_aliasesMem` and `__drv_freesMem` annotations might be needed as well. A false positive is a likely result of a missing `_In_` annotation.

Functions that take a pointer and alias it (thus avoiding a leak) should be annotated with `__drv_aliasesMem`. If you create a function that inserts an object into a global structure, or passes it to a system function that does that, you should add the `__drv_aliasesMem` annotation.

Functions that free memory should be annotated with `__drv_freesMem`. The major functions that free memory already have this annotation.

## Example

The following code example generates this warning:

```
char *p = (char *)malloc(10);
test(p); // does not save a copy of p
```

The following code example avoids this warning:

```
char *p = (char *)malloc(10);
test(p); // does not save a copy of p
free(p);
```

# C28198

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28198: Possibly leaking memory due to an exception.

This warning indicates that allocated memory is not being freed after an exception is raised. The statement at the end of the path can raise an exception. The memory was passed to a function that might have saved a copy to be freed later.

This warning is very similar to warning [C28197](#). The annotations that are recommended for use with warning [C28197](#) can also be used here.

## Example

The following code example generates this warning:

```
char *p1 = new char[10];
char *p2 = new char[10];

test(p1);    // does not save a copy of p

delete[] p2;
delete[] p1;
```

The following code example avoids this warning:

```
char *p1 = new char[10];
char *p2 = NULL;

test(p1);    // does not save a copy of p
try {
    p2 = new char[10];
} catch (std::bad_alloc *e) {
    // just handle the throw
    ;
}
```

# C28199

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28199: Using possibly uninitialized memory

This message indicates that the variable has had its address taken but no assignment to it has been discovered.

The specified variable is being used without being explicitly initialized, but at some point its address was taken, indicating that it might be initialized invisibly to the Code Analysis tool.

This warning can be mistaken if the variable is initialized outside of the function.

The Code Analysis tool reports this warning on function exit if a parameter has an `_Out_` or `_Inout_` annotation and the variable is not initialized.

# C28202

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28202: Illegal reference to non-static member

This warning is reported when there is an error in the annotations.

# C28203

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28203: Ambiguous reference to class member. Could be <name1> or <name2>

This warning is reported when there is an error in the annotations.

# C28204

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28204: <function> : Only one of this overload and the one at <filename>(<line>) are annotated for <paramname>; both or neither must be annotated.

This warning is reported when there is an error in the annotations.

# C28205

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28205: function> : `_Success_` or `_On_failure_` used in an illegal context: <why>

This warning is reported when there is an error in the annotations.

# C28206

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28206: <expression> : left operand points to a struct, use `->`

This warning is reported when the struct pointer dereference notation `->` was expected.

# C28207

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28207: <expression>: left operand is a struct, use .

This warning is reported when a struct dereference dot (.) was expected.

# C28208

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28208: Function <function> was previously defined with a different parameter list at <file>(<line>). Some analysis tools will yield incorrect results

This warning is reported when a function's known definition doesn't match another occurrence.

# C28209

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28209: The declaration for symbol has a conflicting declaration

This warning indicates an incorrectly constructed annotation declaration. This warning should never occur in normal use.

# C28210

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 28210: Annotations for the `_On_failure_` context must not be in explicit pre context

Annotations `_On_failure_` must be explicitly or implicitly indicated in `_post` context, that is, to be applied after the function returns. Use `_drv_out` to ensure this.

# C28211

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28211: Static context name expected for SAL\_context

This warning indicates that the operand to the `_Static_context_` annotation must be the name of a tool-defined context. This warning should never occur in normal use.

# C28212

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28212: Pointer expression expected for annotation

This warning indicates that the numbered parameter to an annotation (not the function being annotated) is expected to be a pointer or array type, but some other type was encountered. The annotation needs to be corrected.

# C28213

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28213: The `_Use_decl_annotations_` annotation must be used to reference, without modification, a prior declaration. <why>

This warning is reported when a prior declaration is referenced without the required `_Use_decl_annotations_` annotation.

# C28214

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28214: Attribute parameter names must be p1...p9

This warning indicates that when you construct an annotation declaration, the parameter names are limited to p1...p9. This warning should never occur in normal use.

# C28215

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28215: The typefix cannot be applied to a parameter that already has a typefix

Applying a `__typefix` annotation to a parameter that already has that annotation is an error. The `__typefix` annotations are used only in a few special cases and this warning is not expected to be seen in normal use.

# C28216

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28216: The `_Check_return_` annotation only applies to post-conditions for the specific function parameter.

The `_Check_return_` annotation has been applied in a context other than `_post`; it may need a `_post` (or `_drv_out`) modifier, or it may be placed incorrectly.

# C28217

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28217: For function, the number of parameters to annotation does not match that found at file

The annotations on the current line and on the line in the message are inconsistent. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in normal use.

# C28218

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28218: For function parameter, the annotation's parameter does not match that found at file

The annotations on the current line and on the line in the message are inconsistent. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in normal use.

# C28219

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28219: Member of enumeration expected for annotation the parameter in the annotation

A parameter to an annotation is expected to be a member of the named `enum` type, and some other symbol was encountered; use a member of that `enum` type. This usually indicates an incorrectly coded annotation macro.

# C28220

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28220: Integer expression expected for annotation the parameter in the annotation

This warning indicates that a parameter to an annotation is expected to be an integer expression, and some other type was encountered. This usually indicates an incorrectly coded annotation macro.

# C28221

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28221: String expression expected for the parameter in the annotation

This warning indicates that a parameter to an annotation is expected to be a string, and some other type was encountered. This usually indicates an incorrectly coded annotation macro and is not expected to be seen in normal use.

# C28222

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 28222: `_Yes_`, `_No_`, or `_Maybe_` expected for annotation

This warning indicates that a parameter to an annotation is expected to be one of the symbols `_Yes_`, `_No_`, or `_Maybe_`, and some other symbol was encountered. This usually indicates an incorrectly coded annotation macro.

# C28223

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning C28223: Did not find expected Token/identifier for annotation, parameter

This warning indicates that a parameter to an annotation is expected to be an identifier, and some other symbol was encountered. This usually indicates an incorrectly coded annotation macro. The use of C or C++ keywords in this position will cause this error.

# C28224

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28224: Annotation requires parameters

This warning indicates that the named annotation is used without parameters and at least one parameter is required. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in normal use.

# C28225

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28225: Did not find the correct number of required parameters in annotation

This warning indicates that the named annotation is used with the incorrect number of parameters. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in typical use.

# C28226

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning C28226: Annotation cannot also be a PrimOp (in current declaration)

This warning indicates that the named annotation is being declared as a PrimOp, and also was previously declared as a normal annotation. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in normal use.

# C28227

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning C28227: Annotation cannot also be a PrimOp (see prior declaration)

This warning indicates that the named annotation is being declared as an ordinary annotation, and also was previously declared as a PrimOp. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in typical use.

# C28228

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28228: Annotation parameter: cannot use type in annotations

This warning indicates that a parameter is of type that is not supported. Annotations can only use a limited set of types as parameters. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in typical use.

# C28229

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28229: Annotation does not support parameters

This warning indicates that an annotation was used with a parameter when the annotation is declared without parameters. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in typical use.

# C28230

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28230: The type of parameter has no member.

This warning indicates that an argument to an annotation attempts to access a `struct`, `class`, or `union` and the named member does not exist.

# C28231

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28231: Annotation is only valid on array

This warning indicates that an argument to an annotation should be an array, and some other type was encountered.

# C28232

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28232: `_Pre_`, `_Post_`, or `_Deref_` not applied to any annotation

This warning indicates that a `_Pre_`, `_Post_`, or `_Deref_` operator appears in an annotation expression without a subsequent functional annotation; the modifier was ignored, but this indicates an incorrectly coded annotation.

# C28233

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28233: pre, post, or deref applied to a block

# C28234

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28234: \_At\_ expression does not apply to current function

This warning indicates that the value of an `_At_` expression does not identify an accessible object.

# C28235

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28235: The function cannot stand alone as an annotation

This warning indicates that an attempt was made to use a function that was not declared to be an annotation in an annotation context. This includes using a primitive operation (PrimOp) in a standalone context. This should not be possible if the standard macros are being used for annotations. This warning is not expected to be seen in typical use.

# C28236

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28236: The annotation cannot be used in an expression

This warning indicates that an attempt was made to use a function declared to be an annotation in an expression context. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in typical use.

# C28237

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28237: The annotation on parameter is no longer supported

An internal error has occurred in the PREfast model file. This warning should not occur in typical use.

# C28238

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28238: The annotation on parameter has more than one of value, stringValue, and longValue. Use paramn=xxx

An internal error has occurred in the PREfast model file. This warning should not occur in typical use.

# C28239

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28239: The annotation on parameter has both value, stringValue, or longValue; and paramn=xxx. Use only paramn=xxx

An internal error has occurred in the PREfast model file. This warning should not occur in typical use.

# C28240

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28240: The annotation on parameter has param2 but no param1

An internal error has occurred in the PREfast model file. This warning should not occur in typical use.

# C28241

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28241: The annotation for function on parameter is not recognized

An unrecognized annotation name was used. This should not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in typical use.

# C28243

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28243: The annotation for function on parameter requires more dereferences than the actual type annotated allows

The number of `__deref` operators on an annotation is more than the number of levels of pointer defined by the parameter type. Correct this warning by changing either the number in the annotation or the pointer levels of the parameter referenced.

# C28244

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28244: The annotation for function has an unparseable parameter/external annotation

This should currently not be possible if the standard macros are being used for annotations; this warning is not expected to be seen in typical use.

# C28245

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28245: The annotation for function annotates 'this' on a non-member-function

An internal error has occurred in the PREfast model file. This warning should not occur in typical use.

# C28246

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28246: The annotation for function '<name>' - parameter '<parameter>' does not match the type of the parameter

A `__deref` operator was applied to a non-pointer type when creating an annotation.

# C28250

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28250: Inconsistent annotation for function: the prior instance has an error.

Note: There are several prototypes for this function. This warning compares the first prototype with instance number <number>.

If a declaration is made using a `typedef`, the line where the `typedef` appears is more useful than the line of the declaration.

This warning refers to an error in the annotation and reflects the requirement that the annotations on a function declaration must match those on the definition, except if a function `typedef` is involved. In that case, the function `typedef` is taken as definitive for both the declaration and the definition.

Note that annotations are usually implemented as macros, and one macro will usually yield several low-level annotations. This warning is reported for each unmatched low-level annotation, so a single unmatched annotation macro may yield a number of unmatched low-level annotations. It is best to simply compare the declaration and definition source code to make sure that they are the same. (Trivial differences in the order of the annotations are not reported.)

The comparison is always between the first declaration found and the current one. If there are additional declarations, each declaration is checked pairwise. It is currently not possible to do a comparison other than in pairs, although it is possible to identify that there are more than two declarations/definitions. The *text* field above contains a list of the annotations that differ (at a fairly low level) between the two instances.

This warning message displays the text of the underlying code sent to the compiler, and not the macros that are used to actually insert the annotation in the source code (as is the case whenever macros are used). In general, you do not need to understand the low-level annotations, but you should recognize that the annotations are being reported as inconsistent between the line numbers reported in the error message. Mostly, an inspection of the source code will make it clear why the inconsistency exists.

# C28251

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28251: Inconsistent annotation for function: this instance has an error

This warning refers to an error in the annotation and reflects the requirement that the annotations on a function declaration must match those on the definition, except if a function `typedef` is involved. In that case, the function `typedef` is taken as definitive for both the declaration and the definition.

Note that annotations are usually implemented as macros, and one macro will usually yield several low-level annotations. This warning is reported for each unmatched low-level annotation, so a single unmatched annotation macro may yield a number of unmatched low-level annotations. It is best to simply compare the declaration and definition source code to make sure that they are the same. (Trivial differences in the order of the annotations are not reported.)

The comparison is always between the first declaration found and the current one. If there are additional declarations, then each declaration is checked in groups of two. It is currently not possible to do a comparison other than in pairs, although it is possible to identify that there are more than two declarations/definitions. The `text` field above contains a list of the annotations that differ (at a fairly low level) between the two instances.

This warning message displays the text of the underlying code sent to the compiler, and not the macros that are used to actually insert the annotation in the source code (as is the case whenever macros are used). In general, you do not need to understand the low-level annotations, but you should recognize that the annotations are being reported as inconsistent between the line numbers reported in the error message. Mostly, an inspection of the source code will make it clear why the inconsistency exists.

# C28252

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28252: Inconsistent annotation for function: parameter has another annotation on this instance

This warning refers to an error in the annotation and reflects the requirement that the annotations on a function declaration must match those on the definition, except if a function `typedef` is involved. In that case, the function `typedef` is taken as definitive for both the declaration and the definition.

Note that annotations are usually implemented as macros, and one macro will usually yield several low-level annotations. This warning is reported for each unmatched low-level annotation, so a single unmatched annotation macro may yield a number of unmatched low-level annotations. It is best to simply compare the declaration and definition source code to make sure that they are the same. (Trivial differences in the order of the annotations are not reported.)

The comparison is always between the first declaration found and the current one. If there are additional declarations, then each declaration is checked in groups of two. It is currently not possible to do a comparison other than in pairs, although it is possible to identify that there are more than two declarations/definitions. The error message contains a list of the annotations that differ (at a fairly low level) between the two instances.

This warning message displays the text of the underlying code sent to the compiler, and not the macros that are used to actually insert the annotation in the source code (as is the case whenever macros are used). In general, you do not need to understand the low-level annotations, but you should recognize that the annotations are being reported as inconsistent between the line numbers reported in the error message. Mostly, an inspection of the source code will make it clear why the inconsistency exists.

# C28253

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28253: Inconsistent annotation for function: parameter has another annotations on this instance

This warning refers to an error in the annotation and reflects the requirement that the annotations on a function declaration must match those on the definition, except if a function `typedef` is involved. In that case, the function `typedef` is taken as definitive for both the declaration and the definition.

Note that annotations are usually implemented as macros, and one macro will usually yield several low-level annotations. This warning is reported for each unmatched low-level annotation, so a single unmatched annotation macro may yield a number of unmatched low-level annotations. It is best to simply compare the declaration and definition source code to make sure that they are the same. (Trivial differences in the order of the annotations are not reported.)

The comparison is always between the first declaration found and the current one. If there are additional declarations, then each declaration is checked in groups of two. It is currently not possible to do a comparison other than in pairs, although it is possible to identify that there are more than two declarations/definitions. The error message contains a list of the annotations that differ (at a fairly low level) between the two instances.

This warning message displays the text of the underlying code sent to the compiler, and not the macros that are used to actually insert the annotation in the source code (as is the case whenever macros are used). In general, you do not need to understand the low-level annotations, but you should recognize that the annotations are being reported as inconsistent between the line numbers reported in the error message. Mostly, an inspection of the source code will make it clear why the inconsistency exists.

# C28254

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28254: `dynamic_cast<>()` is not supported in annotations

The C++ `dynamic_cast` operator cannot be used in annotations.

# C28262

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28262: A syntax error in the annotation was found in function <function> for annotation <name>

# C28263

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28263: A syntax error in a conditional annotation was found for Intrinsic annotation

The Code Analysis tool reports this warning when the return value for the specified function has a conditional value. This warning indicates an error in the annotations, not in the code being analyzed. If the function declaration is in a header file, the annotation should be corrected so that it matches the header file.

The result list for the function and parameter specified has multiple unconditional values.

Typically, this indicates that more than one unconditional `_Null_` or `__drv_valueIs` annotations have been used to specify a result value.

# C28267

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28267: A syntax error in the annotations was found annotation <name> in the function <function>.

# C28272

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28272: The annotation for function, parameter when examining is inconsistent with the function declaration

This warning indicates an error in the annotations, not in the code that is being analyzed. The annotations appearing on a function definition are inconsistent with those appearing on a declaration. The two annotations should be resolved to match.

# C28273

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28273: For function, the clues are inconsistent with the function declaration

This warning indicates an error in the annotations, not in the code that is being analyzed. The annotations appearing on a function definition are inconsistent with those appearing on a declaration. The two annotations should be resolved to match.

# C28275

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28275: The parameter to \_Macro\_value\_ is null

This warning indicates that there is an internal error in the model file, not in the code being analyzed. The *macroValue* function was called without a parameter.

# C28278

2/8/2019 • 2 minutes to read • [Edit Online](#)

Warning C28278: Function name appears with no prototype in scope.

This warning typically indicates that a `__deref` is needed to apply the `__return` annotation to the value returned.

The Code Analysis tool reports this warning when a function without a declaration was called, and the analysis that can be performed is limited because the declaration contains important information.

The C language permits (but discourages) the use of a function for which no prototype has been declared. A function definition or a function declaration (prototype) should appear before the first use of the function. This warning indicates that a function without a declaration was called, and the analysis that can be performed is limited because declaration contains important information. If the function declaration were to contain annotations, the function declaration is even more useful to the Code Analysis tool.

# C28279

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28279: For symbol, a 'begin' was found without a matching 'end'

The annotation language supports a begin and end (`{` and `}` in C) construct, and the pairing has gotten unbalanced. This situation can be avoided if the macros are used.

# C28280

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28280: For symbol, an 'end' was found without a matching 'begin'

The annotation language supports a begin and an end (`{` and `}` in C) construct, and the pairing has gotten unbalanced. This situation can be avoided if the macros are used.

# C28282

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28282: Format Strings must be in preconditions

This warning indicates that a `__drv_formatString` annotation is found, which is not in a `_Pre_` (`__drv_in`) annotation (function parameters are by default `_Pre_`). Check whether the annotation used in an explicit block with a `_Post_` (`__drv_out`) annotation. If so, remove the annotation from any enclosing block that has put it in a `_Post_` context.

# C28283

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28283: For symbol, the specified size specification is not yet supported

The warning indicates that the size specification "sentinel" annotation received a value other than zero. Essentially, the caller tried to say that the string is terminated by a character other than binary zero.

# C28284

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28284: For symbol, Predicates are currently not supported for non-function symbols

This warning indicates that a conditional annotation (predicate, `__drv_when`) was found on something other than a function.

# C28285

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28285: For function, syntax error in parameter

The Code Analysis tool reports this warning when a probable error is encountered in the model file. A few source file errors can also cause such errors.

# C28286

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28286: For function, syntax error near the end

The Code Analysis tool reports this warning when a probable error is encountered in the model file. A few source file errors can also cause such errors.

# C28287

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28287: For function, syntax Error in \_At\_() annotation (unrecognized parameter name)

The Code Analysis tool reports this warning when the `SAL_at` (`__drv_at`) annotation is used and the parameter expression cannot be interpreted in the current context. This might include using a misspelled parameter or member name, or a misspelling of "return" or "this" keywords.

# C28288

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28288: For function, syntax Error in \_At\_() annotation (invalid parameter name)

The Code Analysis tool reports this warning when the `SAL_at` (`__drv_at`) annotation is used and the parameter expression cannot be interpreted in the current context. This might include using a misspelled parameter or member name, or a misspelling of "return" or "this" keywords.

# C28289

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28289: For function: ReadableTo or WritableTo did not have a limit-spec as a parameter

The Code Analysis tool reports this warning when the function/parameter annotation is miscoded as noted.

# C28290

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28290: the annotation for function contains more Externals than the actual number of parameters

The Code Analysis tool reports this warning when the annotation for the function contains more Externals than the actual number of parameters.

# C28291

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28291: Post null/notnull at deref level 0 is meaningless for function <x> at param <number>

The Code Analysis tool reports this warning when the post condition of a dereference level-zero parameter is specified to have a null/non-null property. This error occurs because a value at dereference level zero cannot change.

# C28300

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28300: <parameter\_name>: Expression operands of incompatible types for operator <operator\_name>

This warning is reported when operand types in a parameter are not compatible with the operator.

# C28301

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28301: No annotations for first declaration of <function>. <note1> See <filename>(<line>). <note2>

This warning is reported when annotations were not found at the first declaration of a given function.

# C28302

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28302: For C++ reference-parameter <parameter\_name>, an extra `_Deref_` operator was found on <annotation>.

This warning is reported when an extra level of `_Deref_` is used on a parameter.

SAL2 does not require the use of an extra level of `_Deref_` when dealing with reference parameters. This particular annotation is unambiguous and is interpreted correctly, but should be corrected.

Frequently this can be corrected by simply removing the older `_deref` annotation and using SAL2 syntax. Sometimes may be necessary to use `_At_` to reference the specific object to be annotated.

# C28303

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28303: For C++ reference-parameter <parameter\_name>, an ambiguous `_Deref_` operator was found on <annotation>.

This warning similar to warning C28302 and is reported when an extra level of `_Deref_` is used on a parameter.

SAL2 does not require the use of an extra level of `_Deref_` when dealing with reference parameters. This particular annotation is ambiguous as to which level of dereference is intended to be annotated. It may be necessary to use `_At_` to reference the specific object to be annotated.

## Example

The following code generates this warning because the use of `__deref_out_ecount(n)` is ambiguous:

```
void ref(__deref_out_ecount(n) int **&buff, int &n)
```

The above annotation could be interpreted either as:

- a reference to an array (of n) pointers to integers (SAL1 interpretation)
- a reference to a pointer to an array (of n) integers (SAL2 interpretation)

Either of the following can correct this warning:

```
void ref(_Out_writes_(n) int **&buff, int &n)
// or
_At_(*buff), _Out_writes_(n)) void ref(int **&buff, int &n)
```

# C28304

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28304: For C++ reference-parameter <parameter\_name>, an improperly placed `_Notref_` operator was found applied to <token>.

The `_Notref_` operator should only be used in special circumstances involving C++ reference parameters and only in system-provided macros. It must be immediately followed by a `_Deref_` operator or a functional annotation.

# C28305

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28305: An error while parsing <token> was discovered.

This warning is reported when the expression containing the specified token is ill-formed.

# C28306

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 28306: The annotation on parameter is obsolescent

Use `_string_length_` with the appropriate SAL2 annotation instead.

# C28307

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 28307: The annotation on parameter is obsolescent

Use `_string_length_` with the appropriate SAL2 annotation instead.

# C28308

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 28308: The format list argument position specified by the annotation is incorrect.

The format list argument position must be either a parameter name, or an integer offset that's in the parameter list, or zero.

The second parameter to `IsFormatString2` (`where`) can be in one of two forms:

- A parameter name, which is taken as the first argument to the format string.
- An offset (`n`) relative to the format-string parameter.

In the second form, the first format-string parameter is the `n`-th argument after the format string. If `n` is zero, an ellipsis is specified as the parameter. Specifying an offset of zero without specifying the ellipsis as the first format-string parameter will cause an error.

# C28309

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 28309: <parameter\_name>: Annotation operands must be integer/enum/pointer types. Void operands and C++ overloaded operators are not supported. Floats are approximated as integers. Types: <typelist>.

You've tried to use a void or a function in an annotation expression, and Code Analysis can't handle it. This error typically occurs when an `operator==` that's implemented as a function is used, but other cases may also occur. Examine the types in <typelist> for clues about what's wrong.

# C28310

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 28310: The <annotation\_name> annotation on <function> <parameter> has no SAL version.

All SAL annotations used in source code should have an annotation version applied by the use of SAL\_name. This needs to be corrected in the macro definition.

This warning is reported only once per declaration. Inspect the rest of the declaration for more obsolete SAL.

## See Also

- [Using SAL Annotations to Reduce C/C++ Code Defects](#)

# C28311

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 28311: The <annotation\_name> annotation on <function> <parameter> is an obsolescent version of SAL.

The annotation is an old version and should be updated to the equivalent [SAL2](#). This warning is not emitted if a prior inconsistent annotation warning has been emitted, and is reported only once per declaration. Inspect the rest of the declaration for more obsolete SAL.

## See Also

[Using SAL Annotations to Reduce C/C++ Code Defects](#)

# C28312

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning 28312: The <annotation\_name> annotation on the repeated declaration of <function> <parameter> is an obsolescent version of SAL.

The annotation is an old version and should be updated to the equivalent [SAL2](#). This warning is not emitted if a prior inconsistent annotation warning has been emitted, and is reported only once per declaration. Inspect the rest of the declaration for more obsolete SAL.

## See Also

[Using SAL Annotations to Reduce C/C++ Code Defects](#)

# C28350

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28350: The annotation <annotation> describes a situation that is not conditionally applicable.

Usually this warning is generated when an annotation is applied where the C/C++ type is being inspected.

# C28351

2/8/2019 • 2 minutes to read • [Edit Online](#)

warning C28351: The annotation <annotation> describes where a dynamic value (a variable) cannot be used in the condition.

This warning is reported when an annotation is applied where the C/C++ type is being inspected.

The expression in the `_When_` should evaluate to a constant. The `_When_` is ignored.

# Code analysis application errors

2/8/2019 • 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

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 ' <i>RuleId</i> ' could not be found.
CA0062	The rule ' <i>RuleId</i> ' referenced in rule set ' <i>RuleSetName</i> ' 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 ' <i>TypeName</i> ' contains both a property and a field with the same name ' <i>PropertyFieldName</i> '
CA0066	The value ' <i>VersionID</i> ' provided to the <b>/targetframeworkversion</b> is not a recognized version.

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

## Related sections

- [Code Analysis Policy Errors](#)
- [Analyzing Managed Code Quality](#)

# CA0001

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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 a 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

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

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 all the rule sets included in your rule set exist on disk and that the appropriate rule set directories are being specified in your project through the **CodeAnalysisRuleSetDirectories** property of MSBuild.

To debug the error, examine your rule set file in a text editor. 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 in **Run this rule set**. The Path to the rule set is listed in the description field.

Examine the **Path** attribute value of all the **Include** elements. Include paths can use relative paths to the parent/current rule set file, environment variables, and absolute paths. 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" />
```

Inspect each of these include paths and verify they are all valid.

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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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, also known as noise, or to bad analysis. 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

2/8/2019 • 2 minutes to read • [Edit Online](#)

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

## See Also

[Code Analysis Application Errors](#)

# CA0070

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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

2/8/2019 • 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](#)