Secrets
ABAP
Apex
C
C++
CloudFormation
COBOL
C#
CSS
Flex
Go
HTML
Java
JavaScript
Kotlin
Objective C
PHP
PL/I
PL/SQL
Python
RPG
Ruby
Scala
Swift
Terraform
Text
TypeScript
T-SQL
VB.NET
VB6
XML

C# static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your C# code

All rules 409 | Vulnerability 34 | Bug 76 | Security Hotspot 28 | Code Smell 271 | Quick Fix 52

Tags | Search by name...

emptiness
Code Smell

Boolean literals should not be redundant
Code Smell

Empty statements should be removed
Code Smell

Fields should not have public accessibility
Code Smell

URIs should not be hardcoded
Code Smell

Types should be named in PascalCase
Code Smell

Track uses of "TODO" tags
Code Smell

Classes with "IDisposable" members should implement "IDisposable"
Bug

Calls to "async" methods should not be blocking
Code Smell

Child class fields should not shadow parent class fields
Code Smell

Track lack of copyright and license headers
Code Smell

Exit methods should not be called
Code Smell

Classes should "Dispose" of members from the classes' own "Dispose" methods

Formatting SQL queries is security-sensitive

Analyze your code

Security Hotspot | Major | cwe owasp sans-top25 bad-practice sql

Formatted SQL queries can be difficult to maintain, debug and can increase the risk of SQL injection when concatenating untrusted values into the query. However, this rule doesn't detect SQL injections (unlike rule {rule:csharpsquid:S3649}), the goal is only to highlight complex/formatted queries.

Ask Yourself Whether

- Some parts of the query come from untrusted values (like user inputs).
- The query is repeated/duplicated in other parts of the code.
- The application must support different types of relational databases.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Use parameterized queries, prepared statements, or stored procedures and bind variables to SQL query parameters.
- Consider using ORM frameworks if there is a need to have an abstract layer to access data.

Sensitive Code Example

```
public void Foo(DbContext context, string query, string para
{
    string sensitiveQuery = string.Concat(query, param);
    context.Database.ExecuteSqlCommand(sensitiveQuery); // S
    context.Query<User>().FromSql(sensitiveQuery); // Sensit

    context.Database.ExecuteSqlCommand($"SELECT * FROM mytab
    string query = $"SELECT * FROM mytable WHERE mycol={para
    context.Database.ExecuteSqlCommand(query); // Sensitive,
}

public void Bar(SqlConnection connection, string param)
{
    SqlCommand command;
    string sensitiveQuery = string.Format("INSERT INTO Users
    command = new SqlCommand(sensitiveQuery); // Sensitive

    command.CommandText = sensitiveQuery; // Sensitive

    SqlDataAdapter adapter;
    adapter = new SqlDataAdapter(sensitiveQuery, connection)
}
```

Compliant Solution

Bug

Reading the Standard Input is security-sensitive

Security Hotspot

Using command line arguments is security-sensitive

Security Hotspot

Using Sockets is security-sensitive

Security Hotspot

Encrypting data is security-sensitive

Security Hotspot

```
public void Foo(DbContext context, string query, string para
{
    context.Database.ExecuteSqlCommand("SELECT * FROM mytabl
}
```

See

- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- MITRE, CWE-89 - Improper Neutralization of Special Elements used in an SQL Command
- MITRE, CWE-564 - SQL Injection: Hibernate
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-943 - Improper Neutralization of Special Elements in Data Query Logic
- SANS Top 25 - Insecure Interaction Between Components
- Derived from FindSecBugs rules Potential SQL/JPQL Injection (JPA), Potential SQL/JDOQL Injection (JDO), Potential SQL/HQL Injection (Hibernate)

Available In:

sonarcloud | sonarqube