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...

Vulnerability

Classes should implement their "ExportAttribute" interfaces
Bug

Neither "Thread.Resume" nor "Thread.Suspend" should be used
Bug

"SafeHandle.DangerousGetHandle" should not be called
Bug

Type inheritance should not be recursive
Bug

"IDisposables" should be disposed
Bug

SQL keywords should be delimited by whitespace
Bug

Composite format strings should not lead to unexpected behavior at runtime
Bug

Recursion should not be infinite
Bug

Destructors should not throw exceptions
Bug

Hard-coded credentials are security-sensitive
Security Hotspot

Exceptions should not be thrown from unexpected methods
Code Smell

Database queries should not be vulnerable to injection attacks

Analyze your code

Vulnerability | Blocker | injection cwe owasp sans-top25 sql

User-provided data, such as URL parameters, should always be considered untrusted and tainted. Constructing SQL queries directly from tainted data enables attackers to inject specially crafted values that change the initial meaning of the query itself. Successful database query injection attacks can read, modify, or delete sensitive information from the database and sometimes even shut it down or execute arbitrary operating system commands.

Typically, the solution is to use prepared statements and to bind variables to SQL query parameters with dedicated methods like `setParameter`, which ensures that user-provided data will be properly escaped. Another solution is to validate every parameter used to build the query. This can be achieved by transforming string values to primitive types or by validating them against a white list of accepted values.

Noncompliant Code Example

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using WebApplication1.Controllers;

namespace WebApplicationDotNetCore.Controllers
{
    public class RSPEC3649SQLiNoncompliant : Controller
    {
        private readonly UserAccountContext _context;

        public RSPEC3649SQLiNoncompliant(UserAccountContext
        {
            _context = context;
        }

        public IActionResult Authenticate(string user)
        {
            string query = "SELECT * FROM Users WHERE Userna

            // an attacker can bypass authentication by sett
            // user = "' or 1=1 or ''='";

            var userExists = false;
            if (_context.Database.ExecuteSqlCommand(query) >
            {
                userExists = true;
            }

            return Content(userExists ? "success" : "fail");
        }
    }
}
```

"operator==" should not be overloaded on reference types

Code Smell

Type should not be examined on "System.Type" instances

Code Smell

Test method signatures should be correct

Code Smell

Method overloads with default parameter values should not overlap

Code Smell

Compliant Solution

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using WebApplication1.Controllers;

namespace WebApplicationDotNetCore.Controllers
{
    public class RSPEC3649SQLiCompliant : Controller
    {
        private readonly UserAccountContext _context;

        public RSPEC3649SQLiCompliant(UserAccountContext con
        {
            _context = context;
        }

        public IActionResult Authenticate(string user)
        {
            var query = "SELECT * FROM Users WHERE Username

            var userExists = false;
            if  (_context.Database.ExecuteSqlCommand(query,
            {
                userExists = true;
            }

            return Content(userExists ? "success" : "fail");
        }
    }
}
```

See

- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-89 - Improper Neutralization of Special Elements used in an SQL Command
- MITRE, CWE-564 - SQL Injection: Hibernate
- MITRE, CWE-943 - Improper Neutralization of Special Elements in Data Query Logic
- OWASP SQL Injection Prevention Cheat Sheet
- SANS Top 25 - Insecure Interaction Between Components

Available In:

sonarcloud | sonarqube Developer Edition