

- 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...



members should be named "None"

Code Smell

Overflow checking should not be disabled for "Enumerable.Sum"

Code Smell

Field-like events should not be virtual

Code Smell

Non-constant static fields should not be visible

Code Smell

Inappropriate casts should not be made

Code Smell

Constructors should only call non-overridable methods

Code Smell

"GC.Collect" should not be called

Code Smell

Methods should not be empty

Code Smell

Exceptions should not be thrown in finally blocks

Code Smell

Method overrides should not change parameter defaults

Code Smell

Types allowed to be deserialized should be restricted

Vulnerability

Server-side requests should not be vulnerable to forging attacks

Vulnerability

Members should not have conflicting

Hashes should include an unpredictable salt

Analyze your code

Vulnerability Critical cwe sans-top25 owasp

In cryptography, a "salt" is an extra piece of data which is included when hashing a password. This makes rainbow-table attacks more difficult. Using a cryptographic hash function without an unpredictable salt increases the likelihood that an attacker could successfully find the hash value in databases of precomputed hashes (called rainbow-tables).

This rule raises an issue when a hashing function which has been specifically designed for hashing passwords, such as PBKDF2, is used with a non-random, reused or too short salt value. It does not raise an issue on base hashing algorithms such as sha1 or md5 as they should not be used to hash passwords.

Recommended Secure Coding Practices

- Use hashing functions generating their own secure salt or generate a secure random value of at least 16 bytes.
- The salt should be unique by user password.

Noncompliant Code Example

```
public void Hash(string password)
{
    var salt = Encoding.UTF8.GetBytes("Hardcoded salt");
    var fromHardcoded = new Rfc2898DeriveBytes(password, salt);

    salt = Encoding.UTF8.GetBytes(password);
    var fromPassword = new Rfc2898DeriveBytes(password, salt);

    var shortSalt = new byte[8];
    RandomNumberGenerator.Create().GetBytes(shortSalt);
    var fromShort = new Rfc2898DeriveBytes(password, shortSa
}
```

Compliant Solution





```
public DeriveBytes Hash(string password)
{
    return new Rfc2898DeriveBytes(password, 16);
}
```

See

- [OWASP Top 10 2021 Category A2](#) - Cryptographic Failures
- [OWASP Top 10 2017 Category A3](#) - Sensitive Data Exposure
- [MITRE, CWE-759](#) - Use of a One-Way Hash without a Salt
- [MITRE, CWE-760](#) - Use of a One-Way Hash with a Predictable Salt
- [SANS Top 25](#) - Porous Defenses

Available In:



transparency annotations
 Vulnerability
"PartCreationPolicyAttribute" should be used with "ExportAttribute"
 Bug
"ConstructorArgument" parameters should exist in constructors
 Bug
Windows Forms entry points should be marked with STAThread
 Bug
Collection elements should not be