

- 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...



XML parsers should not be vulnerable to XXE attacks

Vulnerability

A secure password should be used when connecting to a database

Vulnerability

XPath expressions should not be vulnerable to injection attacks

Vulnerability

I/O function calls should not be vulnerable to path injection attacks

Vulnerability

LDAP queries should not be vulnerable to injection attacks

Vulnerability

OS commands should not be vulnerable to command injection attacks

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

"IDisposable" should be disposed

Bug

SQL keywords should be delimited by whitespace

Deserialization should not be vulnerable to injection attacks

Analyze your code

Vulnerability Blocker injection cwe sans-top25 owasp

User-provided data such as URL parameters, POST data payloads or cookies should always be considered untrusted and tainted. Deserialization based on data supplied by the user could result in two types of attacks:

- Remote code execution attacks, where the structure of the serialized data is changed to modify the behavior of the object being unserialized.
- Parameter tampering attacks, where data is modified to escalate privileges or change for example quantity or price of products.

The best way to protect against deserialization attacks is probably to challenge the use of the deserialization mechanism in the application. They are cases where the use of deserialization mechanism was not justified and created breaches (CVE-2017-9785).

If the use of deserialization mechanisms is valid in your context, the problem could be mitigated in any of the following ways:

- Instead of using a native data interchange format, use a safe, standard format such as untyped JSON or structured data approaches such as Google Protocol Buffers.
- To ensure integrity is not compromised, add a digital signature (HMAC) to the serialized data that is validated before deserialization (this is only valid if the client doesn't need to modify the serialized data)
- As a last resort, restrict deserialization to be possible only to specific, whitelisted classes.

Noncompliant Code Example





For `XmlSerializer` serializer, the expected type should not come from user-controlled input:

```
public class XmlSerializerTestCase : Controller
{
    public ActionResult unsecuredeserialization(string typeName)
    {
        // ....
        ExpectedType obj = null;
        Type t = Type.GetType(typeName); // typeName is user-co
        XmlSerializer serializer = new XmlSerializer(t); // Non
        obj = (ExpectedType) serializer.Deserialize(fs);
        // ....
    }
}
```

Compliant Solution

For `XmlSerializer` serializer:

```
public class XmlSerializerTestCase : Controller
{
    public ActionResult securedeserialization()
```

| |
|--|
|  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 |

```
{
    // ....
    ExpectedType obj = null;
    XmlSerializer serializer = new XmlSerializer(typeof(ExpectedType));
    obj = (ExpectedType) serializer.Deserialize(fs);
    // ....
}
```

See

- [OWASP Top 10 2021 Category A8](#) - Software and Data Integrity Failures
- [OWASP Top 10 2017 Category A8](#) - Insecure Deserialization
- [MITRE, CWE-134](#) - Use of Externally-Controlled Format String
- [MITRE, CWE-502](#) - Deserialization of Untrusted Data
- [SANS Top 25](#) - Risky Resource Management

Available In:

  Developer Edition