sonar RULES Products

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

TypeScript static code analysis

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

All rules 279 | Vulnerability 27 | Bug 51 | Security Hotspot 43 | Code Smell 158 | Quick Fix 50

Tags | Search by name...

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

OS commands should not be vulnerable to command injection attacks
Vulnerability

Disabling Vue.js built-in escaping is security-sensitive
Security Hotspot

Disabling Angular built-in sanitization is security-sensitive
Security Hotspot

Hard-coded credentials are security-sensitive
Security Hotspot

Function returns should not be invariant
Code Smell

Assertions should be complete
Code Smell

Tests should include assertions
Code Smell

Octal values should not be used
Code Smell

Switch cases should end with an unconditional "break" statement
Code Smell

"switch" statements should not contain non-case labels
Code Smell

A new session should be created during user authentication
Vulnerability

NoSQL operations should not be vulnerable to injection attacks

Analyze your code

Vulnerability | Blocker | injection cwe owasp sans-top25

User-provided data such as URL parameters and POST body-content should always be considered untrusted and tainted.

Applications that perform NoSQL operations based on tainted data can be exploited similarly to regular SQL injection bugs. Depending on the code, the same risks exist as with SQL injections: The attacker aims to access sensitive information or compromise data integrity. Attacks may involve the injection of query operators, JavaScript code, or string operations.

This problem can be mitigated by using an Object Document Mapper (ODM) library or by validating user-supplied data based on its size or allowed characters.

Since Javascript allows different types of HTTP parameters, the problem could be mitigated by ensuring that the type of the input is a String or by sanitizing the user-provided data.

Noncompliant Code Example

When url query parameters are parsed by the qs module for instance (it's the case by default with express.js framework) then it's possible to inject objects in the URL:

```
function (req, res) {
  let query = { user: req.query.user, city: req.query.city }

  db.collection("users")
    .find(query) // Noncompliant: http://website/?user=admin
    .toArray((err, docs) => { });
}
```

Compliant Solution

Make sure to validate the input types to only handle Strings:

```
function (req, res) {
  let query = { user: req.query.user.toString(), city: req.q

  db.collection("users")
    .find(query) // Compliant
    .toArray((err, docs) => { });
}
```

See

- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- MITRE, CWE-943 - Improper Neutralization of Special Elements in Data Query Logic
- SANS Top 25 - Insecure Interaction Between Components

Available In:

JWT should be signed and verified with strong cipher algorithms

Vulnerability

Cipher algorithms should be robust

Vulnerability

Encryption algorithms should be used with secure mode and padding scheme

Vulnerability

Server hostnames should be verified during SSL/TLS connections

Vulnerability

sonarcloud sonarqube Developer Edition