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...

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

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

Server certificates should be verified during SSL/TLS connections
Vulnerability

Cryptographic keys should be robust
Vulnerability

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

```
var db = require('./mysql/dbConnection.js');

function (req, res) {
  var name = req.query.name; // user-controlled input
  var password = crypto.createHash('sha256').update(req.quer

  var sql = "select * from user where name = '" + name + "'

  db.query(sql, function(err, result) { // Noncompliant
     // something
  })
}
```

Compliant Solution

```
var db = require('./mysql/dbConnection.js');

function (req, res) {
  var name = req.query.name; // user-controlled input
  var password = crypto.createHash('sha256').update(req.quer

  var sql = "select * from user where name = ? and password

  db.query(sql, [name, password], function(err, result) { //
     // something
  })
}
```

See

- OWASP Top 10 2021 Category A3 - Injection

Weak SSL/TLS protocols should not be used

Vulnerability

Origins should be verified during cross-origin communications

Vulnerability

Regular expressions should not be vulnerable to Denial of Service attacks

Vulnerability

File uploads should be restricted

Vulnerability

Regular expressions should be

- 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-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