sonar RULES

Products

# Objective C static code analysis

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

All rules 315 | Vulnerability 10 | Bug 75 | Security Hotspot 18 | Code Smell 212 | Quick Fix 13

Tags

Search by name...

"memset" should not be used to delete sensitive data
Vulnerability

POSIX functions should not be called with arguments that trigger buffer overflows
Vulnerability

Function-like macros should not be invoked without all of their arguments
Bug

The address of an automatic object should not be assigned to another object that may persist after the first object has ceased to exist
Bug

"pthread_mutex_t" should be unlocked in the reverse order they were locked
Bug

"pthread_mutex_t" should be properly initialized and destroyed
Bug

"pthread_mutex_t" should not be consecutively locked or unlocked twice
Bug

Functions with "noreturn" attribute should not return
Bug

"memcmp" should only be called with pointers to trivially copyable types with no padding
Bug

Stack allocated memory and non-owned memory should not be freed
Bug

Closed resources should not be accessed
Bug

Dynamically allocated memory should be released
Bug

## Nested code blocks should not be used

Analyze your code

Code Smell   Minor   bad-practice

Nested code blocks can be used to create a new scope: variables declared within that block cannot be accessed from the outside, and their lifetime end at the end of the block.

While this might seem convenient, using this feature in a function often indicates that it has too many responsibilities and should be refactored into smaller functions.

A nested code block is acceptable when it surrounds all the statements inside an alternative of a `switch` (a `case xxx:` or a `default:`) because it prevents variable declarations from polluting other `cases`.

### Noncompliant Code Example

```
void f(Cache &c, int data) {
  int value;
  { // Noncompliant
    std::scoped_lock l(c.getMutex());
    if (c.hasKey(data)) {
      value = c.get(data);
    } else {
      value = compute(data);
      c.set(data, value);
    }
  } // Releases the mutex

  switch(value) {
    case 1:
    { // Noncompliant, some statements are outside of the blo
       int result = compute(value);
       save(result);
    }
    log();
    break;
    case 2:
    // ...
  }
}
```

### Compliant Solution

```
int getValue(Cache &c, int data) {
  std::scoped_lock l(c.getMutex());
  if (c.hasKey(data)) {
    return c.get(data);
  } else {
    value = compute(data);
    c.set(data, value);
    return value;
  }
}

void f(Cache &c, int data) {
  int value = getValue(c, data);

  switch(value) {
    case 1:
    { // Compliant, limits the scope of "result"
       int result = compute(value);
```

Freed memory should not be used

Bug

Memory locations should not be released more than once

Bug

Memory access should be explicitly bounded to prevent buffer overflows

Bug

Printf-style format strings should not lead to unexpected behavior at runtime

Bug

Recursion should not be infinite

Bug

Resources should be closed

Bug

Hard-coded credentials are security-sensitive

Security Hotspot

"goto" should jump to labels declared later in the same function

Code Smell

Only standard forms of the "defined" directive should be used

Code Smell

Switch labels should not be nested inside non-switch blocks

Code Smell

```
        save(result);
        log();
      }
      break;
      case 2:
      // ...
    }
```

Available In:

sonarcloud sonarqube Developer Edition

© 2008-2022 SonarSource S.A., Switzerland. All content is copyright protected. SONAR, SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource S.A. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved.
Privacy Policy