

- Secrets
- ABAP
- Apex
- C
- C++
- CloudFormation
- COBOL
- C#
- CSS
- Flex
- Go
- HTML
- Java
- JavaScript
- Kotlin
- Kubernetes
- Objective C**
- PHP
- PL/I
- PL/SQL
- Python
- RPG
- Ruby
- Scala
- Swift
- Terraform
- Text
- TypeScript
- T-SQL
- VB.NET
- VB6
- XML



# 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

Functions should not contain too many return statements

Analyze your code

Code Smell Major brain-overload

Having too many return statements in a function increases the function's essential complexity because the flow of execution is broken each time a return statement is encountered. This makes it harder to read and understand the logic of the function.

The way of counting the return statements is aligned with the way we compute **Cognitive Complexity**.

"Under Cyclomatic Complexity, a switch is treated as an analog to an if-else if chain [...] but from a maintainer's point of view, a switch - which compares a single variable to an explicitly named set of literal values - is much easier to understand than an if-else if chain because the latter may make any number of comparisons, using any number of variables and values. "

As a consequence, all the return statements located at the top level of case statements (including default) of a same switch statement count all together as 1.

```
// this counts as 1 return
int fun() {
    switch(variable) {
        case value1:
            return 1;
        case value2:
            return 2;
        default:
            return 3;
    }
}
```

## Noncompliant Code Example

With the default threshold of 3:

```
// this counts as 3 returns
int fun() {
    if (condition1) {
        return 1;
    } else {
        if (condition2) {
            return 0;
        } else {
            return 1;
        }
    }
    return 0;
}
```

```
// this counts as 3 returns
int fun() {
    switch(variable) {
        case value1:
            if(condition1) {
                return 1;
            } else {
                return -1;
            }
        default:
            return 2;
    }
}
```

<b>Freed memory should not be used</b>  Bug
<b>Memory locations should not be released more than once</b>  Bug
<b>Memory access should be explicitly bounded to prevent buffer overflows</b>  Bug
<b>Printf-style format strings should not lead to unexpected behavior at runtime</b>  Bug
<b>Recursion should not be infinite</b>  Bug
<b>Resources should be closed</b>  Bug
<b>Hard-coded credentials are security-sensitive</b>  Security Hotspot
<b>"goto" should jump to labels declared later in the same function</b>  Code Smell
<b>Only standard forms of the "defined" directive should be used</b>  Code Smell
<b>Switch labels should not be nested inside non-switch blocks</b>  Code Smell

```
}

```

Available In:  
**sonarcloud**  | **sonarqube**  Developer Edition