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

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

Analyze your code

Bug Blocker symbolic-execution multi-threading

Mutexes are synchronization primitives that allow to manage concurrency. It is a common situation to have to use multiple *mutexes* to protect multiple resources with different access patterns.

In such a situation, it is crucial to define an order on the set of all *mutexes*.

This order should be strictly followed when *locking mutexes*.

The reverse order should be strictly followed when *unlocking mutexes*.

Failure in doing so can lead to *deadlocks*.

In C++, an easy way to make sure the unlocks are called in reverse order from the lock is to wrap the lock/unlock operations in a RAII class (since destructors of local variables are called in reverse order of their creation).

If instead of `pthread_mutex_t` you are using `std::mutex`, there are other mechanisms that allow you to avoid deadlocks in that case, see {rule:cpp:S5524}.

Noncompliant Code Example

```
pthread_mutex_t mtx1,mtx2;

void bad(void)
{
  pthread_mutex_lock(&mtx1);
  pthread_mutex_lock(&mtx2);
  pthread_mutex_unlock(&mtx1);
  pthread_mutex_unlock(&mtx2);
}
```

Compliant Solution

```
pthread_mutex_t mtx1, mtx2; // if both have to be locked, mtx

void good(void)
{
  pthread_mutex_lock(&mtx1);
  pthread_mutex_lock(&mtx2);
  pthread_mutex_unlock(&mtx2);
  pthread_mutex_unlock(&mtx1);
}
```

Available In:

sonarcloud sonarqube Developer Edition

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