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 properly initialized and destroyed

Analyze your code

Bug Blocker symbolic-execution multi-threading

Mutexes are synchronization primitives that allow to manage concurrency.

Their use requires following a well-defined life-cycle.

- *Mutexes* need to be initialized (`pthread_mutex_init`) before being used. Once it is initialized, a *mutex* is in an *unlocked* state.
- *Mutexes* need to be destroyed (`pthread_mutex_destroy`) to free the associated internal resources. Only *unlocked mutexes* can be safely destroyed.

Before initialization or after destruction, a mutex is in an uninitialized state.

About this life-cycle, the following patterns should be avoided as they result in an undefined behavior:

- trying to initialize an initialized *mutex*
- trying to destroy an initialized *mutex* that is in a *locked* state
- trying to destroy an uninitialized *mutex*
- trying to lock an uninitialized *mutex*
- trying to unlock an uninitialized *mutex*

In C++, it is recommended to wrap mutex creation/destruction in a RAII class, as well as mutex lock/unlock. Those RAII classes will perform the right operations, even in presence of exceptions.

Noncompliant Code Example

```
pthread_mutex_t mtx1;

void bad1(void)
{
  pthread_mutex_init(&mtx1);
  pthread_mutex_init(&mtx1);
}

void bad2(void)
{
  pthread_mutex_init(&mtx1);
  pthread_mutex_lock(&mtx1);
  pthread_mutex_destroy(&mtx1);
}

void bad3(void)
{
  pthread_mutex_init(&mtx1);
  pthread_mutex_destroy(&mtx1);
  pthread_mutex_destroy(&mtx1);
}

void bad4(void)
{
  pthread_mutex_init(&mtx1);
  pthread_mutex_destroy(&mtx1);
  pthread_mutex_lock(&mtx1);
}

void bad5(void)
{
  pthread_mutex_init(&mtx1);
  pthread_mutex_destroy(&mtx1);
  pthread_mutex_unlock(&mtx1);
}
```

Compliant Solution

```
pthread_mutex_t mtx1;

void ok1(void)
{
  pthread_mutex_init(&mtx1);
  pthread_mutex_destroy(&mtx1);
}

void ok2(void)
{
  pthread_mutex_init(&mtx1);
  pthread_mutex_lock(&mtx1);
  pthread_mutex_unlock(&mtx1);
  pthread_mutex_destroy(&mtx1);
}
```

See

- The Open Group pthread_mutex_init, pthread_mutex_destroy

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