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

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

Pointer and reference parameters should be "const" if the corresponding object is not modified

Analyze your code

Code Smell   Minor   Quick Fix   bad-practice misra-c++2008 misra-c2004 misra-c2012

This rule leads to greater precision in the definition of the function interface. The `const` qualification shall be applied to the object pointed to, not to the pointer, since it is the object itself that is being protected.

Noncompliant Code Example

```
void myfunc (      int * param1,  // object is modified
             const int * param2,
                   int * param3, // Noncompliant
                   int * param4) // Noncompliant
{
  *param1 = *param2 + *param3 + *param4;
}

int main (int argc,
          const char * * argv) // Noncompliant
{
  return argc;
}
```

Compliant Solution

```
void myfunc (      int * param1,  // object is modified
             const int * param2,
             const int * param3,
             const int * param4)
{
  *param1 = *param2 + *param3 + *param4;
}

int main (int argc,
          const char * const * argv)
{
  return argc;
}
```

See

- MISRA C:2004, 16.7 - A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
- MISRA C++:2008, 7-1-2 - A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.
- MISRA C:2012, 8.13 - A pointer should point to a const-qualified type whenever possible

Available In:

sonarcloud sonarqube Developer Edition