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

Using "strncpy" or "wcsncpy" is security-sensitive

Analyze your code

Security Hotspot Major cwe owasp cert

In C, a string is just a buffer of characters, normally using the `null` character as a sentinel for the end of the string. This means that the developer has to be aware of low-level details such as buffer sizes or having an extra character to store the final `null` character. Doing that correctly and consistently is notoriously difficult and any error can lead to a security vulnerability, for instance, giving access to sensitive data or allowing arbitrary code execution.

The function `char *strncpy(char * restrict dest, const char * restrict src, size_t count);` copies the first `count` characters from `src` to `dest`, stopping at the first `null` character, and filling extra space with 0. The `wcsncpy` does the same for wide characters and should be used with the same guidelines.

Both of those functions are designed to work with fixed-length strings and might result in a non-`null`-terminated string.

Ask Yourself Whether

- There is a possibility that either the `source` or the `destination` pointer is `null`
- The security of your system can be compromised if the `destination` is a truncated version of the `source`
- The `source` buffer can be both non-`null`-terminated and smaller than the `count`
- The `destination` buffer can be smaller than the `count`
- You expect `dest` to be a `null`-terminated string
- There is an overlap between the `source` and the `destination`

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- C11 provides, in its annex K, the `strncpy_s` and the `wcsncpy_s` that were designed as safer alternatives to `strcpy` and `wcscpy`. It's not recommended to use them in all circumstances, because they introduce a runtime overhead and require to write more code for error handling, but they perform checks that will limit the consequences of calling the function with bad arguments.
- Even if your compiler does not exactly support annex K, you probably have access to similar functions
- If you are using `strncpy` and `wsncpy` as a safer version of `strcpy` and `wcscpy`, you should instead consider `strcpy_s` and `wcscpy_s`, because these functions have several shortcomings:
  - It's not easy to detect truncation
  - Too much work is done to fill the buffer with 0, leading to suboptimal performance
  - Unless manually corrected, the `dest` string might not be `null`-terminated
- If you want to use `strcpy` and `wcscpy` functions and detect if the string was truncated, the pattern is the following:
  - Set the last character of the buffer to `null`
  - Call the function
  - Check if the last character of the buffer is still `null`
- If you are writing C++ code, using `std::string` to manipulate strings is much simpler and less error-prone

Sensitive Code Example

```
int f(char *src) {
  char dest[256];
```

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

```
  strncpy(dest, src, sizeof(dest)); // Sensitive: might silen
  return doSomethingWith(dest);
}
```

Compliant Solution

```
int f(char *src) {
  char dest[256];
  dest[sizeof dest - 1] = 0;
  strncpy(dest, src, sizeof(dest)); // Compliant
  if (dest[sizeof dest - 1] != 0) {
    // Handle error
  }
  return doSomethingWith(dest);
}
```

See

- OWASP Top 10 2021 Category A6 - Vulnerable and Outdated Components
- OWASP Top 10 2017 Category A9 - Using Components with Known Vulnerabilities
- MITRE, CWE-120 - Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- CERT, STR07-C. - Use the bounds-checking interfaces for string manipulation

Available In:

sonarcloud sonarqube Developer Edition