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

C static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your C code

All rules 311 | Vulnerability 13 | Bug 74 | Security Hotspot 18 | Code Smell 206 | Quick Fix 14

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

XML parsers should not be vulnerable to XXE attacks
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

"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);
}
```

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

```
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