

CloudFormation

COBOL

C#

3 CSS

 $\mathbb{X}$ Flex

-GO

Go 5 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



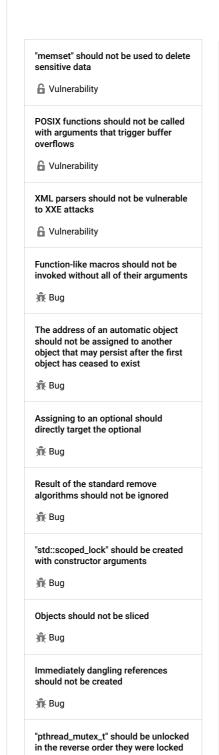
# C++ static code analysis

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

⊗ Code O Quick 68 Fix ΑII 578 Security 18 436 6 Vulnerability (13) **R** Bug (111) rules Hotspot Smell

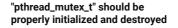
# Bug Blocker

Tags



# Bug

"pthread mutex t" should be properly



Analyze your code

Search by name.

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);
```

### initialized and destroyed

👬 Bug

"pthread\_mutex\_t" should not be consecutively locked or unlocked twice

👬 Bug

"std::move" and "std::forward" should not be confused

🕕 Bug

A call to "wait()" on a "std::condition\_variable" should have a

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

# **Compliant Solution**

```
pthread_mutex_t mtx1;
void okl(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);
```

• The Open Group pthread\_mutex\_init, pthread\_mutex\_destroy

Available In:

sonarlint ⊖ sonarcloud ∴ sonarqube Developer Edition

© 2008-2022 SonarSource S.A., Switzerland. All content is copyright protected. SONAR, SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource S.A. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved.

Privacy Policy