CON53-CPP. Avoid deadlock by locking in a predefined order

Mutexes are used to prevent multiple threads from causing a data race by accessing the same shared resource at the same time. Sometimes, when locking mutexes, multiple threads hold each other’s lock, and the program consequently deadlocks. Four conditions are required for deadlock to occur:

- mutual exclusion (At least one nonshareable resource must be held.),
- hold and wait (A thread must hold a resource while awaiting availability of another resource.),
- no preemption (Resources cannot be taken away from a thread while they are in-use.), and
- circular wait (A thread must await a resource held by another thread which is, in turn, awaiting a resource held by the first thread.).

Deadlock needs all four conditions, so preventing deadlock requires preventing any one of the four conditions. One simple solution is to lock the mutexes in a predefined order, which prevents circular wait.

Noncompliant Code Example

The behavior of this noncompliant code example depends on the runtime environment and the platform’s scheduler. The program is susceptible to deadlock if thread `thr1` attempts to lock `ba2`'s mutex at the same time thread `thr2` attempts to lock `ba1`'s mutex in the `deposit()` function.

```cpp
#include <mutex>
#include <thread>

class BankAccount {
    int balance;
public:
    std::mutex balanceMutex;
    BankAccount() = delete;
    explicit BankAccount(int initialAmount) : balance(initialAmount) {}
    int get_balance() const { return balance; }
    void set_balance(int amount) { balance = amount; }
};

int deposit(BankAccount *from, BankAccount *to, int amount) {
    std::lock_guard<std::mutex> from_lock(from->balanceMutex);
    if (from->get_balance() < amount) {
        return -1; // Indicate error
    }
    std::lock_guard<std::mutex> to_lock(to->balanceMutex);
    from->set_balance(from->get_balance() - amount);
    to->set_balance(to->get_balance() + amount);
    return 0;
}

void f(BankAccount *ba1, BankAccount *ba2) {
    // Perform the deposits.
    std::thread thr1(deposit, ba1, ba2, 100);
    std::thread thr2(deposit, ba2, ba1, 100);
    thr1.join();
    thr2.join();
}
```

Compliant Solution (Manual Ordering)

This compliant solution eliminates the circular wait condition by establishing a predefined order for locking in the `deposit()` function. Each thread will lock on the basis of the `BankAccount` ID, which is set when the `BankAccount` object is initialized.
```cpp
#include <atomic>
#include <mutex>
#include <thread>

class BankAccount {
    static std::atomic<unsigned int> globalId;
    const unsigned int id;
    int balance;

public:
    std::mutex balanceMutex;
    BankAccount() = delete;
    explicit BankAccount(int initialAmount) : id(globalId++), balance(initialAmount) {}
    unsigned int get_id() const { return id; }
    int get_balance() const { return balance; }
    void set_balance(int amount) { balance = amount; }
};

std::atomic<unsigned int> BankAccount::globalId(1);

int deposit(BankAccount *from, BankAccount *to, int amount) {
    std::mutex *first;
    std::mutex *second;
    if (from->get_id() == to->get_id()) {
        return -1; // Indicate error
    }

    // Ensure proper ordering for locking.
    if (from->get_id() < to->get_id()) {
        first = &from->balanceMutex;
        second = &to->balanceMutex;
    } else {
        first = &to->balanceMutex;
        second = &from->balanceMutex;
    }
    std::lock_guard<std::mutex> firstLock(*first);
    std::lock_guard<std::mutex> secondLock(*second);

    // Check for enough balance to transfer.
    if (from->get_balance() >= amount) {
        from->set_balance(from->get_balance() - amount);
        to->set_balance(to->get_balance() + amount);
        return 0;
    }
    return -1;
}

void f(BankAccount *ba1, BankAccount *ba2) {
    // Perform the deposits.
    std::thread thr1(deposit, ba1, ba2, 100);
    std::thread thr2(deposit, ba2, ba1, 100);
    thr1.join();
    thr2.join();
}

Compliant Solution (std::lock())

This compliant solution uses Standard Template Library facilities to ensure that deadlock does not occur due to circular wait conditions. The std::lock() function takes a variable number of lockable objects and attempts to lock them such that deadlock does not occur [ISO/IEC 14882-2014]. In typical implementations, this is done by using a combination of lock(), try_lock(), and unlock() to attempt to lock the object and backing off if the lock is not acquired, which may have worse performance than a solution that locks in predefined order explicitly.
```
```cpp
#include <mutex>
#include <thread>

class BankAccount {
    int balance;
public:
    std::mutex balanceMutex;
    BankAccount() = delete;
    explicit BankAccount(int initialAmount) : balance(initialAmount) {}
    int get_balance() const { return balance; }
    void set_balance(int amount) { balance = amount; }
};

int deposit(BankAccount *from, BankAccount *to, int amount) {
    // Create lock objects but defer locking them until later.
    std::unique_lock<std::mutex> lk1(from->balanceMutex, std::defer_lock);
    std::unique_lock<std::mutex> lk2(to->balanceMutex, std::defer_lock);
    // Lock both of the lock objects simultaneously.
    std::lock(lk1, lk2);
    if (from->get_balance() >= amount) {
        from->set_balance(from->get_balance() - amount);
        to->set_balance(to->get_balance() + amount);
        return 0;
    }
    return -1;
}

void f(BankAccount *ba1, BankAccount *ba2) {
    // Perform the deposits.
    std::thread thr1(deposit, ba1, ba2, 100);
    std::thread thr2(deposit, ba2, ba1, 100);
    thr1.join();
    thr2.join();
}
```

**Risk Assessment**

Deadlock prevents multiple threads from progressing, halting program execution. A denial-of-service attack is possible if the attacker can create the conditions for deadlock.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON53-CPP</td>
<td>Low</td>
<td>Probable</td>
<td>Medium</td>
<td>P4</td>
<td>L3</td>
</tr>
</tbody>
</table>

**Automated Detection**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>CONCURRENCY.LOCK.ORDER</td>
<td>Conflicting lock order</td>
</tr>
<tr>
<td>Coverity</td>
<td>6.5</td>
<td>DEADLOCK</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_CPP-CON53-a</td>
<td>Do not acquire locks in different order</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C++: CON53-CPP</td>
<td>Checks for deadlocks</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>1772, 1773</td>
<td>Enforced by MTA</td>
</tr>
</tbody>
</table>

**Related Vulnerabilities**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

**Related Guidelines**
<table>
<thead>
<tr>
<th>Source</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT Oracle Secure Coding Standard for Java</td>
<td>LCK07-J. Avoid deadlock by requesting and releasing locks in the same order</td>
</tr>
<tr>
<td>SEI CERT C Coding Standard</td>
<td>CON35-C. Avoid deadlock by locking in a predefined order</td>
</tr>
<tr>
<td>MITRE CWE</td>
<td>CWE-764, Multiple Locks of a Critical Resource</td>
</tr>
</tbody>
</table>

**Bibliography**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO/IEC 14882-2014</td>
<td>Subclause 30.4, &quot;Mutual Exclusion&quot;</td>
</tr>
<tr>
<td></td>
<td>Subclause 30.4.3, &quot;Generic Locking Algorithms&quot;</td>
</tr>
</tbody>
</table>