Discussion 5: Focus on C++ - Vulnerabilities in Concurrency

Discussion Topic:

Concurrency in C++ applications can lead to a numberal different vulnerabilities based upon the incorrect implementation of application threads.
What are the best choices to make to avoid common concurrency vulnerabilities and pitfalls?
Be sure to provide an appropriate source code example to illustrate your points.

My Post:

Hello Class,

In computer science, concurrency or multithreading is a powerful that can be used to improve the performance and responsiveness of applications, is especially beneficial to applications needing to handle large amounts of computation or I/O tasks. Concurrency is an approach that allows multiple threads to be executed concurrently on multi-core processors, reducing the program execution time. C++, starting at the C++ 11 release, provides native support for threads such as '*std::thread*', as well as concurrency control features such as '*std::mutex*', '*std::lock_guard*', and '*unique_lock*'.

However, concurrency, in C++, also introduces vulnerabilities if not properly managed. An incorrect implementation of concurrency can result in serious issues such as data races, deadlocks, and undefined behavior. Thus, it is imperative to understand how to properly implement and manage threats to avoid potentially serious consequences. This can be done by adhering to best practices such as the one recommended by SEI CERT C++ Coding Standard, Rule 10 Concurrency (CON) (CMU - Software Engineering Institute, n.d.).

Below is a list of common concurrency vulnerabilities and the corresponding SEI CERT C++ Coding Standard rules that address them.

1. Data Races:
It occurs when two or more threads access shared data simultaneously, and at least one thread modifies the data. This can lead to inconsistent or unexpected results.
Making sure that shared data is accessed in a thread-safe manner is compliant with the SEI CERT C++ Coding Standard ***CON51-CPP****: "Ensure actively held locks are released on exceptional conditions"* (CMU - Software Engineering Institute, n.d.).

Example 1: Data Race (CON51-CPP) – Code with vulnerability

```
int shared_counter = 0;

void increment_counter() {
    for (int i = 0; i < 1000; ++i) {
        ++shared_counter; // Data race here
    }
}

void dataRaceExample() {
    std::thread t1(increment_counter);
    std::thread t2(increment_counter);
    t1.join();
    t2.join();
    std::cout << "Counter: " << shared_counter << std::endl;
}
```

Solution example: Data protected with mutex

```
std::mutex mtx;
int shared_counter = 0;

void increment_counter() {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        ++shared_counter;
    }
}

void dataRaceSolution() {
    std::thread t1(increment_counter);
    std::thread t2(increment_counter);
    t1.join();
    t2.join();
    std::cout << "Counter: " << shared_counter << std::endl;
}
```

2. Deadlocks

It happens when two or more threads are waiting indefinitely for resources held by each other, preventing any of them from proceeding.

Accessing shared data in a safe manner by preventing data races is compliant with the SEI CERT C++ Coding Standard CON51-CPP: "Ensure actively held locks are released on exceptional conditions" (CMU - Software Engineering Institute, n.d.).

Example 2: Deadlock (CON53-CPP) – Code with vulnerability

```
std::mutex mtx1, mtx2;

void thread_func1() { // locks mtx1 waits for mtx2 to be released to lock it
    std::lock_guard<std::mutex> lock1(mtx1);
    // The sleep function allows the other threat to lock the next mutex
    // While this treat is locking the previous mutex
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    std::lock_guard<std::mutex> lock2(mtx2);

    std::cout << "Hello from Threat-1\n";
}

void thread_func2() { // locks mtx2 waits for mtx1 to be released to lock it
    std::lock_guard<std::mutex> lock2(mtx2);
    // The sleep function allows the other threat to lock the next mutex
    // While this treat is locking the previous mutex
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    std::lock_guard<std::mutex> lock1(mtx1);

    std::cout << "Hello from Threat-2\n";
}

void deadlockExample() {
    std::thread t1(thread_func1);
    std::thread t2(thread_func2);
    t1.join();
    t2.join();
}
```

Solution: Lock mutexes in a predefined order

```
std::mutex mtx1, mtx2;

void thread_func1() { // locks mtx1 waits for mtx2 to be released to lock it
    std::lock(mtx1, mtx2); // Locks mutexes in a predefine order
    std::lock_guard<std::mutex> lock1(mtx1, std::adopt_lock);
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    std::lock_guard<std::mutex> lock2(mtx2, std::adopt_lock);

    std::cout << "Hello from Threat-1\n";
}

void thread_func2() { // locks mtx2 waits for mtx1 to be released to lock it
    std::lock(mtx2, mtx1); // Locks mutexes in a predefine order
    std::lock_guard<std::mutex> lock2(mtx2, std::adopt_lock);
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    std::lock_guard<std::mutex> lock1(mtx1, std::adopt_lock);

    std::cout << "Hello from Threat-2\n";
}

void deadlockExample() {
    std::thread t1(thread_func1);
    std::thread t2(thread_func2);
    t1.join();
    t2.join();
}
```

3. Destroying Locked Mutexes

Destroying a mutex while it is still locked can lead to undefined behavior and potential resource leaks. *Ensuring that mutexes are not destroyed while locked is compliant with **CON50-CPP**: "Do not destroy a mutex while it is locked"* (CMU - Software Engineering Institute, n.d.).

Example 3: Destroying locked mutex (CON50-CPP) – Code with vulnerability

```
void mutexDestructionExample() {
    std::mutex mtx;
    mtx.lock();
    // Mutex goes out of scope while still locked
    // causing undefined behavior
}
```

Solution: Unlock Mutex Before Destruction

```
void mutexDestructionSolution() {
    std::mutex mtx;
    mtx.lock();
    // Do work
    mtx.unlock(); // Making sure that the mutex is unlocked
}
```

4. Incorrect Use of Condition Variables

Not wrapping condition variable waits in a loop can lead to spurious wake-ups and incorrect program behavior.

*Wrapping wait calls in a loop is compliant with **CON54-CPP**: "Wrap functions that can spuriously wake up in a loop"* (CMU - Software Engineering Institute, n.d.).

Example 4: Condition variable without loop (CON54-CPP) – Code with vulnerability

```
void consume_list_element() {
    std::unique_lock<std::mutex> lk(m);
    ready = true;

    while(true){
        condition.wait(lk /* The condition predicate is not checked */);

        std::cout << "Hello from Threat!";

        if (ready) { break; }
    }

    // Using implicit loop with lambda predicate
    condition.wait(lk, [] { return ready; });

    std::cout << "Hello from Threat!";
}
```

Solution: Wrap wait in a loop checking predicate

```
static std::mutex m;
static std::condition_variable condition;
bool ready = false;

void consume_list_element() {
    std::unique_lock<std::mutex> lk(m);
    ready = true;

    while(true){

       // The condition predicate is checked
       condition.wait(lk, [] { return ready; });

        std::cout << "Hello from Threat!";

        if (ready) { break; }
    }

    // Using implicit loop with lambda predicate
    condition.wait(lk, [] { return ready; });

    std::cout << "Hello from Threat!";
}
```

5. Speculative Locking of Non-recursive Mutexes

Do not attend to lock a non-recursive mutex that is already locked by the same thread, this can result in undefined behavior and potential deadlocks.

*Avoiding locking non-recursive mutexes is compliant with **CON56-CPP**: "Do not speculatively lock a non-recursive mutex that is already owned by the calling thread"* (CMU - Software Engineering Institute, n.d.)

Example 5: Speculative locking (CON56-CPP) – Code with vulnerability

```
std::mutex mtx; // Not a recursive mutex

void recursiveLock() {
    mtx.lock();
    // Do something
    if (/* some condition */) {
        recursiveLock(); // causes deadlock
    }
    mtx.unlock();
}
```

Solution: Use recursive mutex

```
std::recursive_mutex mtx; // Is a recursive mutex

void recursiveLock() {
    mtx.lock();
    // Do something
    if (/* some condition */) {
        recursiveLock(); // recursive mutex
    }
    mtx.unlock();
}
```

6. Accessing Adjacent Bit-fields from Multiple Threads

Accessing bit fields concurrently can result in data race conditions.
*Preventing data races when accessing bit-fields is compliant with **CON52-CPP:** "Prevent data races when accessing bit-fields from multiple threads"* (CMU - Software Engineering Institute, n.d.).

Example 6: Bit-field Data Race (CON52-CPP) – Code with vulnerability

```
struct Flags {
    unsigned int flag1 : 1;
    unsigned int flag2 : 1;
} flags;

void setFlag1() {
    flags.flag1 = 1;
}

void setFlag2() {
    flags.flag2 = 1;
}
```

Solution: Use separate storage units or protect with mutex

```
std::mutex mtx1, mtx2;
struct Flags {
    unsigned int flag1;
    unsigned int flag2;
} flags;

void setFlag1() {
    std::lock_guard<std::mutex> lock(mtx1);
    flags.flag1 = 1;
}

void setFlag2() {
    std::lock_guard<std::mutex> lock(mtx2);
    flags.flag2 = 1;
}
```

7. Thread Safety and Liveness with Condition Variables

When using condition variables is essential to ensure thread safety and preserve liveness. Liveness is the process of the threads making continuous progress and does not end up in situations where they are unable to proceed. For example, being indefinitely blocked or waiting for resources that never will become available.

*Preserving thread safety and liveness is compliant with **CON55-CPP**: "Preserve thread safety and liveness when using condition variables"* (CMU - Software Engineering Institute, n.d.).

Below is a table summary of the risk assessment of the C++ concurrency coding rules from the SEI CERT C++ Coding Standard.

Table 1: Concurrency Rules Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON50-CPP	Medium	Probable	High	**P4**	**L3**
CON51-CPP	Low	Probable	Low	**P6**	**L2**
CON52-CPP	Medium	Probable	Medium	**P8**	**L2**
CON53-CPP	Low	Probable	Medium	**P4**	**L3**
CON54-CPP	Low	Unlikely	Medium	**P2**	**L3**
CON55-CPP	Low	Unlikely	Medium	**P2**	**L3**
CON56-CPP	Low	Unlikely	High	**P1**	**L3**

Note: From "Rule 10. Concurrency (CON). SEI CERT C++ Coding Standard" by CMU - Software Engineering Institute (n.d.)

Table Description:

- **Rule**: The specific rule from the SEI CERT C++ Coding Standard.
- **Severity**: The potential impact on security.

- **Likelihood**: The probability of the rule violation leading to a vulnerability.
- **Remediation Cost**: The effort required to fix the violation.

To summarize, in computer science, concurrency is a powerful tool that can be used to improve the performance and responsiveness of applications. However, in C++, concurrency can introduce vulnerabilities if not properly managed, and it may result in serious issues such as data races, deadlocks, and undefined behavior. Thus, it is imperative to adhere to best practices such as the one recommended by SEI CERT C++ Coding Standard.-Alex

References:

CMU - Software Engineering Institute (n.d.) Rule 10. Rule 10. Concurrency (CON). *SEI CERT C++ coding standard*. Carnegie Mellon University. Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046460