CON52-CPP. Prevent data races when accessing bit-fields from multiple threads

When accessing a bit-field, a thread may inadvertently access a separate bit-field in adjacent memory. This is because compilers are required to store multiple adjacent bit-fields in one storage unit whenever they fit. Consequently, data races may exist not just on a bit-field accessed by multiple threads but also on other bit-fields sharing the same byte or word. The problem is difficult to diagnose because it may not be obvious that the same memory location is being modified by multiple threads.

One approach for preventing data races in concurrent programming is to use a mutex. When properly observed by all threads, a mutex can provide safe and secure access to a shared object. However, mutexes provide no guarantees with regard to other objects that might be accessed when the mutex is not controlled by the accessing thread. Unfortunately, there is no portable way to determine which adjacent bit-fields may be stored along with the desired bit-field.

Another approach is to insert a non-bit-field member between any two bit-fields to ensure that each bit-field is the only one accessed within its storage unit. This technique effectively guarantees that no two bit-fields are accessed simultaneously.

Noncompliant Code Example (bit-field)

Adjacent bit-fields may be stored in a single memory location. Consequently, modifying adjacent bit-fields in different threads is undefined behavior, as shown in this noncompliant code example.

```
struct MultiThreadedFlags {
  unsigned int flag1 : 2;
  unsigned int flag2 : 2;
};

MultiThreadedFlags flags;

void thread1() {
  flags.flag1 = 1;
}

void thread2() {
  flags.flag2 = 2;
}
```

For example, the following instruction sequence is possible.

```
Thread 1: register 0 = flags
Thread 1: register 0 &= ~mask(flag1)
Thread 2: register 0 = flags
Thread 2: register 0 &= ~mask(flag2)
Thread 1: register 0 |= 1 << shift(flag1)
Thread 1: flags = register 0
Thread 2: register 0 |= 2 << shift(flag2)
Thread 2: flags = register 0
```

Compliant Solution (bit-field, C++11 and later, mutex)

This compliant solution protects all accesses of the flags with a mutex, thereby preventing any data races.

```
#include <mutex>

struct MultiThreadedFlags {
  unsigned int flag1 : 2;
  unsigned int flag2 : 2;
};

struct MtfMutex {
  MultiThreadedFlags s;
  std::mutex mutex;
};

MtfMutex flags;

void thread1() {
  std::lock_guard<std::mutex> lk(flags.mutex);
  flags.s.flag1 = 1;
}

void thread2() {
  std::lock_guard<std::mutex> lk(flags.mutex);
  flags.s.flag2 = 2;
}
```

Compliant Solution (C++11)

In this compliant solution, two threads simultaneously modify two distinct non-bit-field members of a structure. Because the members occupy different bytes in memory, no concurrency protection is required.

```
struct MultiThreadedFlags {
  unsigned char flag1;
  unsigned char flag2;
};

MultiThreadedFlags flags;

void thread1() {
  flags.flag1 = 1;
}

void thread2() {
  flags.flag2 = 2;
}
```

Unlike earlier versions of the standard, C++11 and later explicitly define a memory location and provide the following note in [intro.memory] paragraph 4 [ISO/IEC 14882-2014]:

> *[*Note: *Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of non-zero width.* – end note*]*

It is almost certain that `flag1` and `flag2` are stored in the same word. Using a compiler that conforms to earlier versions of the standard, if both assignments occur on a thread-scheduling interleaving that ends with both stores occurring after one another, it is possible that only one of the flags will be set as intended, and the other flag will contain its previous value because both members are represented by the same word, which is the smallest unit the processor can work on. Before the changes made to the C++ Standard for C++11, there were no guarantees that these flags could be modified concurrently.

Risk Assessment

Although the race window is narrow, an assignment or an expression can evaluate improperly because of misinterpreted data resulting in a corrupted running state or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level

CON52-CPP	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
Axivion Bauhaus Suite	6.9.0	**CertC++-CON52**	
Coverity	6.5	**RACE_CONDITION**	Fully implemented
Parasoft C/C++test	10.4.2	**CERT_CPP-CON52-a**	Use locks to prevent race conditions when modifying bit fields
Polyspace Bug Finder	R2019b	CERT C++: CON52-CPP	Checks for data races (rule partially covered)
PRQA QA-C++	4.4	**1774, 1775**	Enforced by MTA

Related Vulnerabilities

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

Related Guidelines

SEI CERT C Coding Standard	CON32-C. Prevent data races when accessing bit-fields from multiple threads

Bibliography

[ISO/IEC 14882-2014]	Subclause 1.7, "The C++ memory model"