CON32-C. 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. A similar problem is discussed in CON43-C. Do not allow data races in multithreaded code, but the issue described by this rule can be harder 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 multi_threaded_flags {
  unsigned int flag1 : 2;
  unsigned int flag2 : 2;
};

struct multi_threaded_flags flags;

int thread1(void *arg) {
  flags.flag1 = 1;
  return 0;
}

int thread2(void *arg) {
  flags.flag2 = 2;
  return 0;
}
```

The C Standard, 3.14, paragraph 3 [ISO/IEC 9899:2011], states

> *NOTE 2 A bit-field and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields, if one is declared inside a nested structure 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 member declaration. It is not safe to concurrently update two non-atomic bit-fields in the same structure if all members declared between them are also (non-zero-length) bit-fields, no matter what the sizes of those intervening bit-fields happen to be.*

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, C11, Mutex)

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

```
#include <threads.h>

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

struct mtf_mutex {
  struct multi_threaded_flags s;
  mtx_t mutex;
};

struct mtf_mutex flags;

int thread1(void *arg) {
  if (thrd_success != mtx_lock(&flags.mutex)) {
    /* Handle error */
  }
  flags.s.flag1 = 1;
  if (thrd_success != mtx_unlock(&flags.mutex)) {
    /* Handle error */
  }
  return 0;
}

int thread2(void *arg) {
  if (thrd_success != mtx_lock(&flags.mutex)) {
    /* Handle error */
  }
  flags.s.flag2 = 2;
  if (thrd_success != mtx_unlock(&flags.mutex)) {
    /* Handle error */
  }
  return 0;
}
```

Compliant Solution (C11)

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 multi_threaded_flags {
  unsigned char flag1;
  unsigned char flag2;
};

struct multi_threaded_flags flags;

int thread1(void *arg) {
  flags.flag1 = 1;
  return 0;
}

int thread2(void *arg) {
  flags.flag2 = 2;
  return 0;
}
```

Unlike C99, C11 explicitly defines a memory location and provides the following note in subclause 3.14.2 [ISO/IEC 9899:2011]:

> *NOTE 1 Two threads of execution can update and access separate memory locations without interfering with each other.*

It is almost certain that `flag1` and `flag2` are stored in the same word. Using a compiler that conforms to C99 or earlier, 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. 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 were made to the C Standard for C11, 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
CON32-C	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**read_data_race** **write_data_race**	Supported by sound analysis (data race alarm)
Axivion Bauhaus Suite	6.9.0	**CertC-CON32**	
CodeSonar	5.1p0	**CONCURRENCY.DATARACE**	Data race
Coverity	2017.07	**MISSING_LOCK**	Partially implemented
Parasoft C/C++test	10.4.2	**CERT_C-CON32-a**	Use locks to prevent race conditions when modifying bit fields
Polyspace Bug Finder	R2019b	CERT C: Rule CON32-C	Checks for data race (rule fully covered)

Related Vulnerabilities

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

Bibliography

[ISO/IEC 9899:2011]	3.14, "Memory Location"