POS49-C. When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed

When multiple threads must access or make modifications to a common variable, they may also inadvertently access other variables adjacent in memory. This is an artifact of variables being stored compactly, with one byte possibly holding multiple variables, and is a common optimization on word-addressed machines. Bit-fields are especially prone to this behavior because compliers are allowed to store multiple bit-fields in one addressable byte or word. This implies that race conditions may exist not just on a variable accessed by multiple threads but also on other variables sharing the same byte or word address. This recommendation is a specific instance of CON32-C. Prevent data races when accessing bit-fields from multiple threads using POSIX threads.

A common tool for preventing race conditions in concurrent programming is the mutex. When properly observed by all threads, a mutex can provide safe and secure access to a common variable; however, it guarantees nothing with regard to other variables that might be accessed when a common variable is accessed.

Unfortunately, there is no portable way to determine which adjacent variables may be stored along with a certain variable.

A better approach is to embed a concurrently accessed variable inside a union, along with a `long` variable, or at least some padding to ensure that the concurrent variable is the only element to be accessed at that address. This technique would effectively guarantee that no other variables are accessed or modified when the concurrent variable is accessed or modified.

Noncompliant Code Example (Bit-field)

In this noncompliant code example, two executing threads simultaneously access two separate members of a global `struct`:

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

struct multi_threaded_flags flags;

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

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

Although this code appears to be harmless, it is likely that `flag1` and `flag2` are stored in the same byte. 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 equal its previous value, because both bit-fields are represented by the same byte, which is the smallest unit the processor can work on.

For example, the following sequence of events can occur:

```
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
```

Even though each thread is modifying a separate bit-field, they are both modifying the same location in memory. This is the same problem discussed in CON43-C. Do not allow data races in multithreaded code but is harder to diagnose because it is not immediately obvious that the same memory location is being modified.

Compliant Solution (Bit-field)

This compliant solution protects all accesses of the flags with a mutex, thereby preventing any thread-scheduling interleaving from occurring. In addition, the flags are declared `volatile` to ensure that the compiler will not attempt to move operations on them outside the mutex. Finally, the flags are embedded in a union alongside a `long`, and a static assertion guarantees that the flags do not occupy more space than the `long`. This technique prevents any data not checked by the mutex from being accessed or modified with the bit-fields.

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

union mtf_protect {
  struct multi_threaded_flags s;
  long padding;
};

static_assert(sizeof(long) >= sizeof(struct multi_threaded_flags));

struct mtf_mutex {
  union mtf_protect u;
  pthread_mutex_t mutex;
};

struct mtf_mutex flags;

void thread1(void) {
  int result;
  if ((result = pthread_mutex_lock(&flags.mutex)) != 0) {
    /* Handle error */
  }
  flags.u.s.flag1 = 1;
  if ((result = pthread_mutex_unlock(&flags.mutex)) != 0) {
    /* Handle error */
  }
}

void thread2(void) {
  int result;
  if ((result = pthread_mutex_lock(&flags.mutex)) != 0) {
    /* Handle error */
  }
  flags.u.s.flag2 = 2;
  if ((result = pthread_mutex_unlock(&flags.mutex)) != 0) {
    /* Handle error */
  }
}
```

Static assertions are discussed in detail in DCL03-C. Use a static assertion to test the value of a constant expression.

Risk Assessment

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS49-C	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
Axivion Bauhaus Suite	6.9.0	**CertC-POS49**	
CodeSonar	5.2p0	**CONCURRENCY.DATARACE**	Data race
Coverity	2017.07	**Various concurrency checkers**	Partially implemented; needs further investigation
Parasoft C/C++test	10.4.2	**CERT_C-POS49-a**	Use locks to prevent race conditions when modifying bit fields
Polyspace Bug Finder	R2019b	CERT C: Rule POS49-C	Checks for data race (rule partially covered)
PRQA QA-C	9.7	**1774,1775**	Enforced by MTA

Related Vulnerabilities

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

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.7.2.1, "Structure and Union Specifiers"