CON08-C. Do not assume that a group of calls to independently atomic methods is atomic

A consistent locking policy guarantees that multiple threads cannot simultaneously access or modify shared data. When two or more operations must be performed as a single atomic operation, a consistent locking policy must be implemented using some form of locking, such as a mutex. In the absence of such a policy, the code is susceptible to race conditions.

When presented with a set of operations, where each is guaranteed to be atomic, it is tempting to assume that a single operation consisting of individually-atomic operations is guaranteed to be collectively atomic without additional locking. A grouping of calls to such methods requires additional synchronization for the group.

Compound operations on shared variables are also non-atomic. See CON07-C. Ensure that compound operations on shared variables are atomic for more information.

Noncompliant Code Example

This noncompliant code example stores two integers atomically. It also provides atomic methods to obtain their sum and product. All methods are locked with the same mutex to provide their atomicity.

```
#include <threads.h>
#include <stdio.h>
#include <stdbool.h>

static int a = 0;
static int b = 0;
mtx_t lock;

bool init_mutex(int type) {
  /* Validate type */
  if (thrd_success != mtx_init(&lock, type)) {
    return false;  /* Report error */
  }
  return true;
}

void set_values(int new_a, int new_b) {
  if (thrd_success != mtx_lock(&lock)) {
    /* Handle error */
  }
  a = new_a;
  b = new_b;
  if (thrd_success != mtx_unlock(&lock)) {
    /* Handle error */
  }
}

int get_sum(void) {
  if (thrd_success != mtx_lock(&lock)) {
    /* Handle error */
  }
  int sum = a + b;
  if (thrd_success != mtx_unlock(&lock)) {
    /* Handle error */
  }
  return sum;
}

int get_product(void) {
  if (thrd_success != mtx_lock(&lock)) {
    /* Handle error */
  }
  int product = a * b;
  if (thrd_success != mtx_unlock(&lock)) {
    /* Handle error */
  }
  return product;
}

/* Can be called by multiple threads */
void multiply_monomials(int x1, int x2) {
  printf("(x + %d)(x + %d)\n", x1, x2);
  set_values( x1, x2);
  printf("= x^2 + %dx + %d\n", get_sum(), get_product());
}
```

Unfortunately, the `multiply_monomials()` function is still subject to race conditions, despite relying exclusively on atomic function calls. It is quite possible for `get_sum()` and `get_product()` to work with different numbers than the ones that were set by `set_values()`. It is even possible for `get_sum()` to operate with different numbers than `get_product()`.

Compliant Solution

This compliant solution locks the `multiply_monomials()` function with the same mutex lock that is used by the other functions. For this code to work, the mutex must be recursive. This is accomplished by making it recursive in the `init_mutex()` function.

```
#include <threads.h>
#include <stdio.h>
#include <stdbool.h>

extern void set_values(int, int);
extern int get_sum(void);
extern int get_product(void);

mtx_t lock;

bool init_mutex(int type) {
  /* Validate type */
  if (thrd_success != mtx_init(&lock, type | mtx_recursive)) {
    return false;  /* Report error */
  }
  return true;
}

/* Can be called by multiple threads */
void multiply_monomials(int x1, int x2) {
  if (thrd_success != mtx_lock(&lock)) {
    /* Handle error */
  }
  set_values( x1, x2);
  int sum = get_sum();
  int product = get_product();
  if (thrd_success != mtx_unlock(&lock)) {
    /* Handle error */
  }

  printf("(x + %d)(x + %d)\n", x1, x2);
  printf("= x^2 + %dx + %d\n", sum, product);
}
```

Noncompliant Code Example

Function chaining is a useful design pattern for building an object and setting its optional fields. The output of one function serves as an argument (typically the last) in the next function. However, if accessed concurrently, a thread may observe shared fields to contain inconsistent values. This noncompliant code example demonstrates a race condition that can occur when multiple threads can variables with no thread protection.

```
#include <threads.h>
#include <stdio.h>

typedef struct currency_s {
  int quarters;
  int dimes;
  int nickels;
  int pennies;
} currency_t;

currency_t *set_quarters(int quantity, currency_t *currency) {
  currency->quarters += quantity;
  return currency;
}
currency_t *set_dimes(int quantity, currency_t *currency) {
  currency->dimes += quantity;
  return currency;
}
currency_t *set_nickels(int quantity, currency_t *currency) {
  currency->nickels += quantity;
  return currency;
}
currency_t *set_pennies(int quantity, currency_t *currency) {
  currency->pennies += quantity;
  return currency;
}

int init_45_cents(void *currency) {
  currency_t *c = set_quarters(1, set_dimes(2, currency));
  /* Validate values are correct */
  return 0;
}
int init_60_cents(void* currency) {
  currency_t *c = set_quarters(2, set_dimes(1, currency));
  /* Validate values are correct */
  return 0;
}

int main(void) {
  thrd_t thrd1;
  thrd_t thrd2;
  currency_t currency = {0, 0, 0, 0};

  if (thrd_success != thrd_create(&thrd1, init_45_cents, &currency)) {
    /* Handle error */
  }
  if (thrd_success != thrd_create(&thrd2, init_60_cents, &currency)) {
    /* Handle error */
  }
  if (thrd_success != thrd_join(thrd1, NULL)) {
    /* Handle error */
  }
  if (thrd_success != thrd_join(thrd2, NULL)) {
    /* Handle error */
  }

  printf("%d quarters, %d dimes, %d nickels, %d pennies\n",
         currency.quarters, currency.dimes, currency.nickels, currency.pennies);
  return 0;
}
```

In this noncompliant code example, the program constructs a currency `struct` and starts two threads that use method chaining to set the optional values of the structure. This example code might result in the currency `struct` being left in an inconsistent state, for example, with two quarters and one dime or one quarter and two dimes.

Noncompliant Code Example

This code remains unsafe even if it uses a mutex on the `set` functions to guard modification of the currency:

```
#include <threads.h>
#include <stdio.h>

typedef struct currency_s {
  int quarters;
  int dimes;
  int nickels;
  int pennies;
  mtx_t lock;
} currency_t;

currency_t *set_quarters(int quantity, currency_t *currency) {
  if (thrd_success != mtx_lock(&currency->lock)) {
    /* Handle error */
  }
  currency->quarters += quantity;
  if (thrd_success != mtx_unlock(&currency->lock)) {
    /* Handle error */
  }
  return currency;
}
currency_t *set_dimes(int quantity, currency_t *currency) {
  if (thrd_success != mtx_lock(&currency->lock)) {
    /* Handle error */
  }
  currency->dimes += quantity;
  if (thrd_success != mtx_unlock(&currency->lock)) {
    /* Handle error */
  }
  return currency;
}
currency_t *set_nickels(int quantity, currency_t *currency) {
  if (thrd_success != mtx_lock(&currency->lock)) {
    /* Handle error */
  }
  currency->nickels += quantity;
  if (thrd_success != mtx_unlock(&currency->lock)) {
    /* Handle error */
  }
  return currency;
}
currency_t *set_pennies(int quantity, currency_t *currency) {
  if (thrd_success != mtx_lock(&currency->lock)) {
    /* Handle error */
  }
  currency->pennies += quantity;
  if (thrd_success != mtx_unlock(&currency->lock)) {
    /* Handle error */
  }
  return currency;
}

int init_45_cents(void *currency) {
  currency_t *c = set_quarters(1, set_dimes(2, currency));
  /* Validate values are correct */
  return 0;
}
int init_60_cents(void* currency) {
  currency_t *c = set_quarters(2, set_dimes(1, currency));
  /* Validate values are correct */
  return 0;
}

int main(void) {
  int result;
  thrd_t thrd1;
  thrd_t thrd2;
  currency_t currency = {0, 0, 0, 0};

  if (thrd_success != mtx_init(&currency.lock, mtx_plain)) {
    /* Handle error */
```

```
  }
  if (thrd_success != thrd_create(&thrd1, init_45_cents, &currency)) {
    /* Handle error */
  }
  if (thrd_success != thrd_create(&thrd2, init_60_cents, &currency)) {
    /* Handle error */
  }

  if (thrd_success != thrd_join(thrd1, NULL)) {
    /* Handle error */
  }
  if (thrd_success != thrd_join(thrd2, NULL)) {
    /* Handle error */
  }

  printf("%d quarters, %d dimes, %d nickels, %d pennies\n",
         currency.quarters, currency.dimes, currency.nickels, currency.pennies);

  mtx_destroy( &currency.lock);
  return 0;
}
```

Compliant Solution

This compliant solution uses a mutex, but instead of guarding the `set` functions, it guards the `init` functions, which are invoked at thread creation.

```
#include <threads.h>
#include <stdio.h>
typedef struct currency_s {
  int quarters;
  int dimes;
  int nickels;
  int pennies;
  mtx_t lock;
} currency_t;

currency_t *set_quarters(int quantity, currency_t *currency) {
  currency->quarters += quantity;
  return currency;
}
currency_t *set_dimes(int quantity, currency_t *currency) {
  currency->dimes += quantity;
  return currency;
}
currency_t *set_nickels(int quantity, currency_t *currency) {
  currency->nickels += quantity;
  return currency;
}
currency_t *set_pennies(int quantity, currency_t *currency) {
  currency->pennies += quantity;
  return currency;
}

int init_45_cents(void *currency) {
  currency_t *c = (currency_t *)currency;
  if (thrd_success != mtx_lock(&c->lock)) {
    /* Handle error */
  }
  set_quarters(1, set_dimes(2, currency));
  if (thrd_success != mtx_unlock(&c->lock)) {
    /* Handle error */
  }
  return 0;
}
int init_60_cents(void *currency) {
  currency_t *c = (currency_t *)currency;
  if (thrd_success != mtx_lock(&c->lock)) {
    /* Handle error */
  }
```

```
  set_quarters(2, set_dimes(1, currency));
  if (thrd_success != mtx_unlock(&c->lock)) {
    /* Handle error */
  }
  return 0;
}

int main(void) {
  int result;
  thrd_t thrd1;
  thrd_t thrd2;
  currency_t currency = {0, 0, 0, 0};

  if (thrd_success != mtx_init(&currency.lock, mtx_plain)) {
    /* Handle error */
  }
  if (thrd_success != thrd_create(&thrd1, init_45_cents, &currency)) {
    /* Handle error */
  }
  if (thrd_success != thrd_create(&thrd2, init_60_cents, &currency)) {
    /* Handle error */
  }

  if (thrd_success != thrd_join(thrd1, NULL)) {
    /* Handle error */
  }
  if (thrd_success != thrd_join(thrd2, NULL)) {
    /* Handle error */
  }

  printf("%d quarters, %d dimes, %d nickels, %d pennies\n",
         currency.quarters, currency.dimes, currency.nickels, currency.pennies);

  mtx_destroy(&currency.lock);
  return 0;
}
```

Consequently this compliant solution is thread-safe, and will always print out the same number of quarters as dimes.

Risk Assessment

Failure to ensure the atomicity of two or more operations that must be performed as a single atomic operation can result in race conditions in multithreaded applications.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON08-C	Low	Probable	Medium	**P4**	**L3**

Related Guidelines

CERT Oracle Secure Coding Standard for Java	VNA03-J. Do not assume that a group of calls to independently atomic methods is atomic VNA04-J. Ensure that calls to chained methods are atomic
MITRE CWE	CWE-362, Concurrent execution using shared resource with improper synchronization ("race condition") CWE-366, Race condition within a thread CWE-662, Improper synchronization

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.26, "Threads <threads.h>"