CON43-C. Do not allow data races in multithreaded code

When multiple threads can read or modify the same data, use synchronization techniques to avoid software flaws that can lead to security vulnerabilities. Data races can often result in abnormal termination or denial of service, but it is possible for them to result in more serious vulnerabilities. The C Standard, section 5.1.2.4, paragraph 25 [ISO/IEC 9899:2011], says:

> *The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.*

Noncompliant Code Example

Assume this simplified code is part of a multithreaded bank system. Threads call `credit()` and `debit()` as money is deposited into and withdrawn from the single account. Because the addition and subtraction operations are not atomic, it is possible that two operations can occur concurrently, but only the result of one would be saved—despite declaring the `account_balance` `volatile`. For example, an attacker can credit the account with a sum of money and make a large number of small debits concurrently. Some of the debits might not affect the account balance because of the race condition, so the attacker is effectively creating money.

```
static volatile int account_balance;

void debit(int amount) {
  account_balance -= amount;
}

void credit(int amount) {
  account_balance += amount;
}
```

Compliant Solution (Mutex)

This compliant solution uses a mutex to make credits and debits atomic operations. All credits and debits will now affect the account balance, so an attacker cannot exploit the race condition to steal money from the bank. The mutex is created with the `mtx_init()` function. The presence of the mutex makes declaring `account_balance` `volatile` unnecessary.

```
#include <threads.h>

static int account_balance;
static mtx_t account_lock;

int debit(int amount) {
  if (mtx_lock(&account_lock) == thrd_error) {
    return -1;   /* Indicate error to caller */
  }
  account_balance -= amount;
  if (mtx_unlock(&account_lock) == thrd_error) {
    return -1;   /* Indicate error to caller */
  }
  return 0;   /* Indicate success */
}

int credit(int amount) {
  if (mtx_lock(&account_lock) == thrd_error) {
    return -1;   /* Indicate error to caller */
  }
  account_balance += amount;
  if (mtx_unlock(&account_lock) == thrd_error) {
    return -1;   /* Indicate error to caller */
  }
  return 0;   /* Indicate success */
}

int main(void) {
  if(mtx_init(&account_lock, mtx_plain) == thrd_error) {
    /* Handle error */
  }
  /* ... */
}
```

Compliant Solution (Atomic)

This compliant solution uses an atomic variable to synchronize credit and debit operations. All credits and debits will now affect the account balance, so an attacker cannot exploit the race condition to steal money from the bank. The atomic integer does not need to be initialized because default (zero) initialization of an atomic object with static or thread-local storage is guaranteed to produce a valid state. The `+=` and `-=` operators behave atomically when used with an atomic variable.

```
#include <stdatomic.h>

atomic_int account_balance;

void debit(int amount) {
  account_balance -= amount;
}

void credit(int amount) {
  account_balance += amount;
}
```

Noncompliant Code Example (Double-Fetch)

This noncompliant code example illustrates Xen Security Advisory CVE-2015-8550 / XSA-155 In this example, the following code is vulnerable to a data race where the integer referenced by `ps` could be modified by a second thread that ran between the two reads of the variable.

```
#include <stdio.h>

void doStuff(int *ps) {
  switch (*ps) {
    case 0: { printf("0"); break; }
    case 1: { printf("1"); break; }
    case 2: { printf("2"); break; }
    case 3: { printf("3"); break; }
    case 4: { printf("4"); break; }
    default: { printf("default"); break; }
  }
}
```

Even though there is only one read of the `*ps` variable in the source code, the compiler is permitted to produce object code that performs multiple reads of the memory location. This is permitted by the "as-if" principle, as explained by section 5.1 of the [C99 Rationale 2003]:

> *The /as if/ principle is invoked repeatedly in this Rationale. The C89 Committee found that describing various aspects of the C language, library, and environment in terms of concrete models best serves discussion and presentation. Every attempt has been made to craft these models so that implementations are constrained only insofar as they must bring about the same result, /as if/ they had implemented the presentation model; often enough the clearest model would make for the worst implementation.*

Implementation Details (GCC)

This code produces two reads of the `*ps` value using GCC 4.8.4 on x86, as well as GCC 5.3.0 on x86-64 (Compiler-Introduced Double-Fetch Vulnerabilities – Understanding XSA-155).

Noncompliant Code Example (Volatile)

The data race can be disabled by declaring the data to be volatile, because the `volatile` keyword forces the compiler to not produce two reads of the data. However, this violates CON02-C. Do not use volatile as a synchronization primitive.

```
#include <stdio.h>

void doStuff(volatile int *ps) {
  switch (*ps) {
    case 0: { printf("0"); break; }
    case 1: { printf("1"); break; }
    case 2: { printf("2"); break; }
    case 3: { printf("3"); break; }
    case 4: { printf("4"); break; }
    default: { printf("default"); break; }
  }
}
```

Compliant Solution (C11, Atomic)

Declaring the data to be atomic also forces the compiler to produce only one read of the data.

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

void doStuff(atomic_int *ps) {
  switch (atomic_load(ps)) {
    case 0: { printf("0"); break; }
    case 1: { printf("1"); break; }
    case 2: { printf("2"); break; }
    case 3: { printf("3"); break; }
    case 4: { printf("4"); break; }
    default: { printf("default"); break; }
  }
}
```

Compliant Solution (C11, Fences)

The bug was actually resolved by erecting fences around the `switch` statement.

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

void doStuff(int *ps) {
  atomic_thread_fence(memory_order_acquire);
  switch (*ps) {
    case 0: { printf("0"); break; }
    case 1: { printf("1"); break; }
    case 2: { printf("2"); break; }
    case 3: { printf("3"); break; }
    case 4: { printf("4"); break; }
    default: { printf("default"); break; }
  }
  atomic_thread_fence(memory_order_release);
}
```

Risk Assessment

Race conditions caused by multiple threads concurrently accessing and modifying the same data can lead to abnormal termination and denial-of-service attacks or data integrity violations.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
CON43-C	Medium	Probable	High	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**read_data_race** **write_data_race**	Supported by sound analysis (data race alarm)
CodeSonar	5.1p0	**CONCURRENCY.DATARACE**	Data race
Coverity	2017.07	**MISSING_LOCK (partial)**	Implemented
Parasoft C/C++test	10.4.2	**CERT_C-CON43-a**	Usage of functions prone to race is not allowed
Polyspace Bug Finder	R2019a	CERT C: Rule CON43-C	Checks for data race (rule fully covered)

Related Vulnerabilities

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

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CWE 2.11	CWE-366, Race condition within a thread	2017-07-07: CERT: Exact

Bibliography

[ISO/IEC 9899:2011]	5.1.2.4, "Multi-threaded Executions and Data Races" 7.17.2, "Initialization"
[C99 Rationale 2003]	
[Dowd 2006]	Chapter 13, "Synchronization and State"

[Plum 2012]	
[Seacord 2013]	Chapter 8, "File I/O"