CON35-C. Avoid deadlock by locking in a predefined order

Mutexes are used to prevent multiple threads from causing a data race by accessing shared resources at the same time. Sometimes, when locking mutexes, multiple threads hold each other's lock, and the program consequently deadlocks. Four conditions are required for deadlock to occur:

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Deadlock needs all four conditions, so preventing deadlock requires preventing any one of the four conditions. One simple solution is to lock the mutexes in a predefined order, which prevents circular wait.

Noncompliant Code Example

The behavior of this noncompliant code example depends on the runtime environment and the platform's scheduler. The program is susceptible to deadlock if thread `thr1` attempts to lock `ba2`'s mutex at the same time thread `thr2` attempts to lock `ba1`'s mutex in the `deposit()` function.

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

typedef struct {
  int balance;
  mtx_t balance_mutex;
} bank_account;

typedef struct {
  bank_account *from;
  bank_account *to;
  int amount;
} transaction;

void create_bank_account(bank_account **ba,
                         int initial_amount) {
  bank_account *nba = (bank_account *)malloc(
    sizeof(bank_account)
  );
  if (nba == NULL) {
    /* Handle error */
  }

  nba->balance = initial_amount;
  if (thrd_success
      != mtx_init(&nba->balance_mutex, mtx_plain)) {
    /* Handle error */
  }

  *ba = nba;
}

int deposit(void *ptr) {
  transaction *args = (transaction *)ptr;

  if (thrd_success != mtx_lock(&args->from->balance_mutex)) {
    /* Handle error */
  }

  /* Not enough balance to transfer */
  if (args->from->balance < args->amount) {
    if (thrd_success
        != mtx_unlock(&args->from->balance_mutex)) {
      /* Handle error */
    }
    return -1; /* Indicate error */
  }
  if (thrd_success != mtx_lock(&args->to->balance_mutex)) {
    /* Handle error */
  }

  args->from->balance -= args->amount;
```

```
  args->to->balance += args->amount;

  if (thrd_success
      != mtx_unlock(&args->from->balance_mutex)) {
    /* Handle error */
  }

  if (thrd_success
      != mtx_unlock(&args->to->balance_mutex)) {
    /* Handle error */
  }

  free(ptr);
  return 0;
}

int main(void) {
  thrd_t thr1, thr2;
  transaction *arg1;
  transaction *arg2;
  bank_account *ba1;
  bank_account *ba2;

  create_bank_account(&ba1, 1000);
  create_bank_account(&ba2, 1000);

  arg1 = (transaction *)malloc(sizeof(transaction));
  if (arg1 == NULL) {
    /* Handle error */
  }
  arg2 = (transaction *)malloc(sizeof(transaction));
  if (arg2 == NULL) {
    /* Handle error */
  }
  arg1->from = ba1;
  arg1->to = ba2;
  arg1->amount = 100;

  arg2->from = ba2;
  arg2->to = ba1;
  arg2->amount = 100;

  /* Perform the deposits */
  if (thrd_success
     != thrd_create(&thr1, deposit, (void *)arg1)) {
    /* Handle error */
  }
  if (thrd_success
      != thrd_create(&thr2, deposit, (void *)arg2)) {
    /* Handle error */
  }
  return 0;
}
```

Compliant Solution

This compliant solution eliminates the circular wait condition by establishing a predefined order for locking in the `deposit()` function. Each thread will lock on the basis of the `bank_account` ID, which is set when the `bank_account struct` is initialized.

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

typedef struct {
  int balance;
  mtx_t balance_mutex;

  /* Should not change after initialization */
  unsigned int id;
} bank_account;
```

```
typedef struct {
  bank_account *from;
  bank_account *to;
  int amount;
} transaction;

unsigned int global_id = 1;

void create_bank_account(bank_account **ba,
                         int initial_amount) {
  bank_account *nba = (bank_account *)malloc(
    sizeof(bank_account)
  );
  if (nba == NULL) {
    /* Handle error */
  }

  nba->balance = initial_amount;
  if (thrd_success
      != mtx_init(&nba->balance_mutex, mtx_plain)) {
    /* Handle error */
  }

  nba->id = global_id++;
  *ba = nba;
}

int deposit(void *ptr) {
  transaction *args = (transaction *)ptr;
  int result = -1;
  mtx_t *first;
  mtx_t *second;

  if (args->from->id == args->to->id) {
    return -1; /* Indicate error */
  }

  /* Ensure proper ordering for locking */
  if (args->from->id < args->to->id) {
    first = &args->from->balance_mutex;
    second = &args->to->balance_mutex;
  } else {
    first = &args->to->balance_mutex;
    second = &args->from->balance_mutex;
  }
  if (thrd_success != mtx_lock(first)) {
    /* Handle error */
  }
  if (thrd_success != mtx_lock(second)) {
    /* Handle error */
  }

  /* Not enough balance to transfer */
  if (args->from->balance >= args->amount) {
    args->from->balance -= args->amount;
    args->to->balance += args->amount;
    result = 0;
  }

  if (thrd_success != mtx_unlock(second)) {
    /* Handle error */
  }
  if (thrd_success != mtx_unlock(first)) {
    /* Handle error */
  }
  free(ptr);
  return result;
}
```

Risk Assessment

Deadlock prevents multiple threads from progressing, halting program execution. A denial-of-service attack is possible if the attacker can create the conditions for deadlock.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON35-C	Low	Probable	Medium	**P4**	**L3**

Related Vulnerabilities

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

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**deadlock**	Supported by sound analysis (deadlock alarm)
CodeSonar	5.1p0	**CONCURRENCY.LOCK.ORDER**	Conflicting lock order
Coverity	2017.07	**ORDER_REVERSAL**	Fully implemented
Klocwork	2018	**CONC.DL**	
Parasoft C/C++test	10.4.2	**CERT_C-CON35-a**	Avoid double locking
Polyspace Bug Finder	R2019b	CERT C: Rule CON35-C	Checks for deadlock (rule fully covered)
PRQA QA-C	9.7	**1772,1773**	Enforced by MTA

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT Oracle Secure Coding Standard for Java	LCK07-J. Avoid deadlock by requesting and releasing locks in the same order	Prior to 2018-01-12: CERT: Unspecified Relationship