Module 2: Program Security (Attacks)
data races

Meng Xu  (University of Waterloo)
Spring 2023
Outline

1. Why studying data races?
2. Intuitive definition
3. Formal reasoning
4. Data race vs atomicity
5. Other form of races
What is data race?

```
global var count = 0
global var mutex = ⊥
```

```
for(i = 0; i < x; i++) {
    /* do sth critical */
    .......
    lock(mutex);
    count++;
    unlock(mutex);
}
```

```
for(i = 0; i < y; i++) {
    /* do sth critical */
    .......
    lock(mutex);
    count++;
    unlock(mutex);
}
```

Thread 1

Thread 2

**Q:** What is the value of `count` when both threads terminate?
Data races are not tied to a specific programming language, instead, they are tied to data sharing in concurrent execution.

For example, in the database context:

**Q:** If two database clients send the following requests concurrently, what will be the result (both try to withdraw $100 from Alice)?

**Client 1**
```
SELECT @balance = Balance
    FROM Ledger WHERE Name = "Alice";

UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
```

**Client 2**
```
SELECT @balance = Balance
    FROM Ledger WHERE Name = "Alice";

UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
```
Data race in a database setting

**One possible interleaving (that messes up the states)**

```
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
```

**Q:** How to prevent data race in this case?

**Interleavings with transactions**

```
BEGIN TRANSACTION;
    SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
    UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
COMMIT TRANSACTION;
BEGIN TRANSACTION;
    SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
    UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
COMMIT TRANSACTION;
```
Recall the “nice” properties of memory error

Data race is a common attack vector and building blocks for sophisticated exploitations... just like memory error.

- Memory errors have universally accepted definitions
  - Once you find a memory error, you do not need to diligently argue that this is a bug and not a feature
- Memory errors often lead to a set of known consequences that are generally considered severe (e.g., data leak or denial-of-service)
  - Once you find a memory error, you do not need to construct a working exploit to justify it
- Finding memory errors typically do not require program-specific domain knowledge
  - If you have a technique that can find memory errors in one codebase, you can scale it up to millions of codebases

In fact, very few types of vulnerabilities meet these requirements. data race is one of them!
Data races have *universally* accepted definitions
- Once you find a data race, you do not need to diligently argue that this is a bug and not a feature

Data races often lead to a set of known consequences that are *generally* considered severe (e.g., data leak or denial-of-service)
- Once you find a data race, you do not need to construct a working exploit to justify it

Finding data races typically *do not require* program-specific domain knowledge
- If you have a technique that can find data races in one codebase, you can scale it up to millions of codebases

Data races can only happen in programs with data sharing through a concurrency model, e.g., multi-threaded or distributed programs.
Data race may lead to memory errors

p is a global pointer initialized to NULL

```c
if (!p) {
    p = malloc(128);
}
if (p) {
    free(p);
    p = NULL;
}
```

Thread 1

```c
if (!p) {
    p = malloc(256);
}
if (p) {
    free(p);
    p = NULL;
}
```

Thread 2

Q: What are the possible outcomes of this execution?
Data race as heisenbug

Programs which contain data races usually demonstrate unexpected and even non-deterministic behavior.

- The outcome might depend on a specific execution order (a.k.a. thread interleaving).
- Re-running the program may not always produce the same results.

Concurrent programs are hard to debug and even harder to ensure correctness.
Outline

1. Why studying data races?

2. Intuitive definition

3. Formal reasoning

4. Data race vs atomicity

5. Other form of races
An intuitive definition

Intuitively, a *data race* happens when:

1. There are two memory accesses from *different threads*.
2. Both accesses target the *same memory location*.
3. At least one of them is a *write* operation.
4. Both accesses could *interleave* freely without restrictions such as *synchronization primitives* or *causality relations*.
Data race definition in C++ standard

When

- an evaluation of an expression writes to a memory location and
- another evaluation reads or modifies the same memory location,
the expressions are said to conflict.

A program that has two conflicting evaluations has a data race unless:

- both evaluations execute on the same thread, or
- both conflicting evaluations are atomic operations, or
- one of the conflicting evaluations happens-before another.

Adapted from a community-backed C++ reference site. For the full version, please refer to the related sections in C++ working draft.
Revisit the example

global var count = 0

---

for(i = 0; i < x; i++) {
    count++;
}

Thread 1

---

for(i = 0; i < y; i++) {
    count++;
}

Thread 2
Free interleavings without locking
Revisit the example

```c
// Intuitive Formulation

global var count = 0

for(i = 0; i < x; i++) {
    lock(mutex);
    count++;
    unlock(mutex);
}

Thread 1

for(i = 0; i < y; i++) {
    lock(mutex);
    count++;
    unlock(mutex);
}

Thread 2
```
Limited interleavings with locking
Common synchronization primitives

- Lock / Mutex / Critical section
- Read-write lock
- Barrier
- Semaphore
Revisiting the definition

Intuitively, a *data race* happens when:

1. There are two memory accesses from different threads.
2. Both accesses target the same memory location.
3. At least one of them is a *write* operation.
4. Both accesses could *interleave* freely without restrictions such as synchronization primitives or *causality relations*. 
# Causality relations: an example

```c
#include <stdio.h>
#include <pthread.h>

int i;
int retval;

void* foo(void* p) {
    printf("Value of i: %d\n", i);
    printf("Value of j: %d\n", *(int*)p);
    pthread_exit(&retval);
}

int main(void) {
    int i = 1;
    int j = 2;
    pthread_t id;
    pthread_create(&id, NULL, foo, &j);
    pthread_join(id, NULL);
    printf("Return value from thread: %d\n", retval);
}
```
Causality relations

Thread 1

`var i`  
`var j`

`pthread_create`

`<thread start>`

`var i`  
`var j`

`var retval`

`<thread end>`

Thread 2

`var retval`

`pthread_join`
<table>
<thead>
<tr>
<th></th>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Why studying data races?</td>
</tr>
<tr>
<td>2</td>
<td>Intuitive definition</td>
</tr>
<tr>
<td>3</td>
<td>Formal reasoning</td>
</tr>
<tr>
<td>4</td>
<td>Data race vs atomicity</td>
</tr>
<tr>
<td>5</td>
<td>Other form of races</td>
</tr>
</tbody>
</table>
Revisiting the definition

If we can find, statically or dynamically, a pair of memory access instructions \((A_1, A_2)\) such that

- they originate from different threads,
- both \(A_1\) and \(A_2\) target the same memory location, \textit{AND}
- at least one of them is a write operation,

then we conclude that \((A_1, A_2)\) must be one of the following cases:

1. \(A_1\) strictly happens before \(A_2\) or vice versa due to causality, \textit{OR}
2. \(A_1\) and \(A_2\) can only occur when a common lock is held, \textit{OR}
3. \((A_1, A_2)\) is a data race.

\textbf{Q:} Wait... how are locks implemented?
How are synchronization primitives implemented?

- **Hardware support**
  - Atomic swap
  - Atomic read-modify-write
    * compare-and-swap
    * test-and-set
    * fetch-and-add
    * ......

- **Software algorithms**
  - Dekker’s algorithm
Spinlock with atomic swap (\texttt{xchg})

```assembly
locked:                        ; The lock variable. 1 = locked, 0 = unlocked.
    dd 0

spin_lock:
    mov eax, 1                   ; Set the EAX register to 1.
    xchg eax, [locked]          ; Atomically swap the EAX register with
                                ; the lock variable.
                                ; This will always store 1 to the lock, leaving
                                ; the previous value in the EAX register.
    test eax, eax                ; Test EAX with itself. Among other things, this
                                ; will set the processor’s Zero Flag if EAX is 0.
                                ; If EAX is 0, then the lock was unlocked and
                                ; we just locked it.
                                ; Otherwise, EAX is 1 and we didn’t acquire the lock.
    jnz spin_lock                ; Jump back to the MOV instruction if the Zero Flag is
                                ; not set; the lock was previously locked, and so
                                ; we need to spin until it becomes unlocked.
    ret                          ; The lock has been acquired, return to the caller.

spin_unlock:
    xor eax, eax                 ; Set the EAX register to 0.
    xchg eax, [locked]          ; Atomically swap the EAX register with
                                ; the lock variable.
    ret                          ; The lock has been released.
```


### Dekker’s algorithm

```c
1 bool wants_to_enter[2] = {false, false};
2 int turn = 0;  // or turn = 1 */

1 // lock
2 wants_to_enter[0] = true;
3 while (wants_to_enter[1]) {
  4   if (turn != 0) {
      5      wants_to_enter[0] = false;
      6      // busy wait
      7      while (turn != 0) {}  
      8      wants_to_enter[0] = true;
  9  }
10 }
11  /* ... critical section ... */
12 // unlock
13 turn = 1;
14 wants_to_enter[0] = false;

1 // lock
2 wants_to_enter[1] = true;
3 while (wants_to_enter[0]) {
  4   if (turn != 1) {
      5      wants_to_enter[1] = false;
      6      // busy wait
      7      while (turn != 1) {}  
      8      wants_to_enter[1] = true;
  9  }
10 }
11  /* ... critical section ... */
12 // unlock
13 turn = 0;
14 wants_to_enter[1] = false;
```

Thread 1

Thread 2
Dekker’s algorithm

**Q:** Suppose that you are not aware that Dekker’s algorithm is implementing a lock, are there data races in Dekker’s algorithm?

**A:** By looking at the code, yes... However, this is often called a benign data race.
Is this a data race?

```
int x = 0;
bool flag = false;
lock mutex = unlocked;
```

Thread 1

```
x++;
lock(mutex);
flag = true;
unlock(mutex);
```

Thread 2

```
while(true) {
lock(mutex);
if (flag) {
    break;
}
unlock(mutex);
x--;
```
Is this a data race?

```c
1 int x = 0;
2 bool flag = false;

1 x++; 
2 flag = true;

Thread 1

1 while (!flag) {};
2 x--; 

Thread 2
```
How to model concurrency mathematically?

- Lamport clock
- Vector clock
Lamport clock algorithm

Each thread has its own clock variable $t$

- **On initialization:**
  - $t \leftarrow 0$
- **On write to shared memory $*\text{ptr} = \text{val}$:**
  - $t \leftarrow t + 1$
  - store $t$ alongside $\text{val}$ at memory location $\text{ptr}$
- **On read from shared memory $\text{val} = *\text{ptr}$:**
  - retrieve the stored clock $t'$ at memory location $\text{ptr}$
  - $t \leftarrow \max(t, t') + 1$

**Properties of Lamport clock:**

- $a \rightarrow b \implies L(a) < L(b)$
- $L(a) < L(b) \iff a \rightarrow b$
Introduction
Intuitive
Formal
Atomicity
Other

## Vector clock algorithm

Each thread $i$ has its own clock vector $t$

- **On initialization:**
  - $T \leftarrow \langle 0, 0, \ldots, 0 \rangle_N$, assuming $N$ threads

- **On write to shared memory $\*\text{ptr} = \text{val}$:**
  - $T[i] \leftarrow T[i] + 1$
  - store $T$ alongside $\text{val}$ at memory location $\text{ptr}$

- **On read from shared memory $\text{val} = \*\text{ptr}$:**
  - retrieve the stored clock $T'$ at memory location $\text{ptr}$
  - $\forall k \in [0, N) : T[k] = \max(T[k], T'[k])$
  - $T[i] \leftarrow T[i] + 1$
Properties of the vector clock algorithm

With the following definition on the timestamp ordering:

- $T = T'$ $\iff$ $\forall i \in [0, N): T[i] = T'[i]$
- $T \leq T'$ $\iff$ $\forall i \in [0, N): T[i] \leq T'[i]$
- $T < T'$ $\iff$ $T \leq T' \land T \neq T'$
- $T \parallel T'$ $\iff$ $T \not\leq T' \land T' \not\leq T$

We have:

- $a \rightarrow b$ $\iff$ $V(a) < V(b)$
- $a = b$ $\iff$ $V(a) = V(b)$
- $a \parallel b$ $\iff$ $V(a) \parallel V(b)$
Practice exercise (vector clock)

1 int x = 0;
2 bool flag = false;

1 x++;
2 flag = true;

1 while (!flag) {};
2 x--; 

Thread 1

Thread 2

Prove: the write of x at x-- in thread 2 can never happen before the read of x in x++ in thread 1.
Practice exercise (vector clock)

1 int x = 0;
2 bool r = false;

1 v = load(&x);
2 store(&x, v + 1);
3 store(&r, true);

1 loop:
2 c = load(&r);
3 jump_if_false(c, loop);
4 v = load(&x);
5 store(&x, v - 1);

Thread 1

Thread 2

Prove: line 5 at thread 2 can never happen before line 1 at thread 1.
Outline

1. Why studying data races?
2. Intuitive definition
3. Formal reasoning
4. Data race vs atomicity
5. Other form of races
Revisit the example

global var `count = 0`

---

**Thread 1**

```cpp
for(i = 0; i < x; i++) {
    lock(mutex);
    t = count;
    unlock(mutex);
    t++;
    lock(mutex);
    count = t;
    unlock(mutex);
}
```

**Thread 2**

```cpp
for(i = 0; i < y; i++) {
    lock(mutex);
    t = count;
    unlock(mutex);
    t++;
    lock(mutex);
    count = t;
    unlock(mutex);
}
```
Revisit the example

**Q:** In this modified example, is there a data race?

**A:** No

**Q:** But the results are the same with all locks removed?

```
global var count = 0

for(i = 0; i < x; i++) {
    t = count;
    t++;
    count = t;
}
```

```
for(i = 0; i < y; i++) {
    t = count;
    t++;
    count = t;
}
```

**A:** No, depending on how hardware works (e.g., per-bit conflict)
Q: What is common in developers’ expectations in the two variants?

A: State do not change for a critical section during execution.

A: **Generalization**: state remain integral for a critical section during execution. No change of states is just one way of remaining integral (assuming state is integral before the critical section).
State integrity example

```
struct R { x: int, y: int } g;
[invariant] g.x + g.y == 100;

int add_x(v: int) {
  g.x += v;
  g.y -= v;
}

Thread 1

int add_y(v: int) {
  g.y += v;
  g.x -= v;
}

Thread 2
```
State integrity example

The code snippet below demonstrates a state integrity example using a struct to represent a state with two integer fields, `x` and `y`, and an invariant that their sum should always be 100. The code includes two threads, each updating the state with a function that adds an integer value to one of the fields, while ensuring the invariant is maintained through locking mechanisms.

```c
struct R { x: int, y: int } g;
[invariant] g.x + g.y == 100;
lock mutex = unlocked;

int add_x(v: int) {
lock(mutex);
g.x += v;
unlock(mutex);
lock(mutex);
g.y -= v;
unlock(mutex);
}

int add_y(v: int) {
lock(mutex);
g.y += v;
unlock(mutex);
lock(mutex);
g.x -= v;
unlock(mutex);
}
```

**Q:** Is this the right way of adding locks?

**A:** No, as the invariant is not guaranteed.
State integrity example

```c
1 struct R { x: int, y: int } g;
2 [invariant] g.x + g.y == 100;
3 lock mutex = unlocked;
```

```c
1 int add_x(v: int) {
2   lock(mutex);
3   g.x += v;
4   g.y -= v;
5   unlock(mutex);
6 }
```

```c
1 int add_y(v: int) {
2   lock(mutex);
3   g.y += v;
4   g.x -= v;
5   unlock(mutex);
6 }
```

**Q:** Is this the right way of adding locks?

**A:** Yes, the invariant is guaranteed at each entry and exit of the critical section in both threads.
State integrity is hard to capture

However, in practice, the invariant often exists in

- some architectural design documents (which no one reads)
- code comments in a different file (which no one notices)
- folklore knowledge among the dev team
- the mind of the developer who has resigned a few years ago...
Outline

1. Why studying data races?
2. Intuitive definition
3. Formal reasoning
4. Data race vs atomicity
5. Other form of races
A more abstract view of data race

Q: Why data race can happen in the first place?

A: Because two threads in the same process share memory

We can further generalize this concept by asking:

Q: What else do they share?
Q: What about other entities that may run concurrently?

And the answer to these questions will help define race condition.
Example: race over the filesystem

```c
#include <...>

int main(int argc, char *argv[]) {
    FILE *fd;
    struct stat buf;

    if (stat("/some_file", &buf)) {
        exit(1); // cannot read stat message
    }

    if (buf.st_uid != getuid()) {
        exit(2); // permission denied
    }

    fd = fopen("/some_file", "wb+");
    if (fd == NULL) {
        exit(3); // unable to open the file
    }

    fprintf(f, "<some-secret-value>");
    fclose(fd);
    return 0;
}
```
Example: the Dirty COW exploit

CVE-2016-5195

Allows local privilege escalation: user(1000) → root(0).

Exists in the kernel for nine years before finally patched.

Details on the Website.
⟨ End ⟩