ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis

SIGMOD 2017

Presentation: Dimitrios Stamelos

Motivation

Concurrency bugs can become security vulnerabilities

- Database transaction assumed to ensure correctness
- In practice, concurrent execution may break assumptions
- Attackers can exploit this via API calls
- Example:

  Balance = 100$
  Two withdrawals of 99$ -> Final balance = -98$

Real-World Attack

Flexcoin Bitcoin Exchange (2014)

- Attacker sent thousands of concurrent requests
- Exploited race condition in transfer logic
- Funds were transferred before balances updated
- Entire platform shut down

flexcoin the bitcoin bank

FAQ Bitcoin Bank Register Activate Login

Flexcoin is shutting down. (March 3 2014)

On March 2nd 2014 Flexcoin was attacked and robbed of all coins in the hot wallet. The attacker made off with 896 BTC, dividing them into these two addresses:

1NDkevapt4SWYFEmquCDBSf7DLMTNVggdu

1QFcC5JitGwpFKqRDd9QNH3eGN56dCNgy6

Update (March 4 2014)

During the investigation into stolen funds we have determined that the extent of the theft was enabled by a flaw within the front-end.

The attacker logged into the flexcoin front end from IP address 207.12.89.117 under a newly created username and deposited to address 1DSD3B3uS2wGZjZAwa2dqQ7M9v7Ajw2iLy

"By sending thousands of simultaneous requests, the attacker was able to 'move' coins from one user account to another until the sending account was overdrawn, before balances were updated."

Target Problem

Transactions are assumed to guarantee consistency

- In reality:
  - Weak isolation levels(e.g. Read Committed)
  - Incorrect transaction usage
  - High concurrency via web APIs

- So non-serializable behavior may occur

ACIDRain Attack

Exploiting concurrency anomalies by issuing concurrent API requests

- No access to database or server required
- Only uses public APIs (HTTP/REST)
- Triggers non-serializable behavior
- All of the above leads to incorrect application state (business rules)
- ACID:
  -A: Atomicity
  -C: Consistency
  -I: Isolation
  -D: Durability

Two Types of Anomalies

Source of Vulnerabilities

1. Level-Based Anomalies
   - Caused by weak database isolation
   - Example: Lost Update

2. Scope-Based Anomalies
   - Incorrect transaction boundaries
   - Multiple operations not encapsulated

Key finding:

Most vulnerabilities are scope-based

Voucher Example

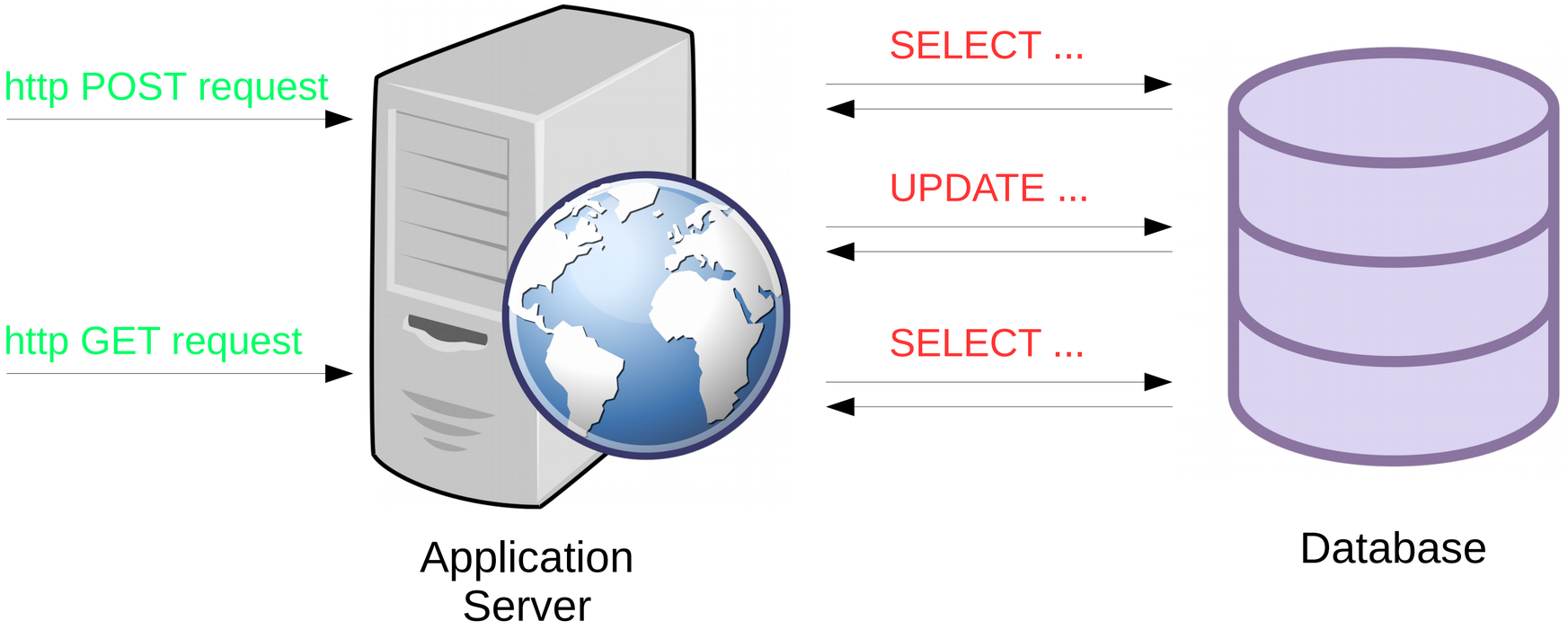
Voucher should be used once

- Application logic:
  - Read voucher usage
  - If usage == 0 -> mark as used

- Under concurrency:
  - Two requests read usage = 0
  - Both pass the check
  - Both update

-> This gives us a result of voucher used multiple times

Problem Setup: Attacking Websites



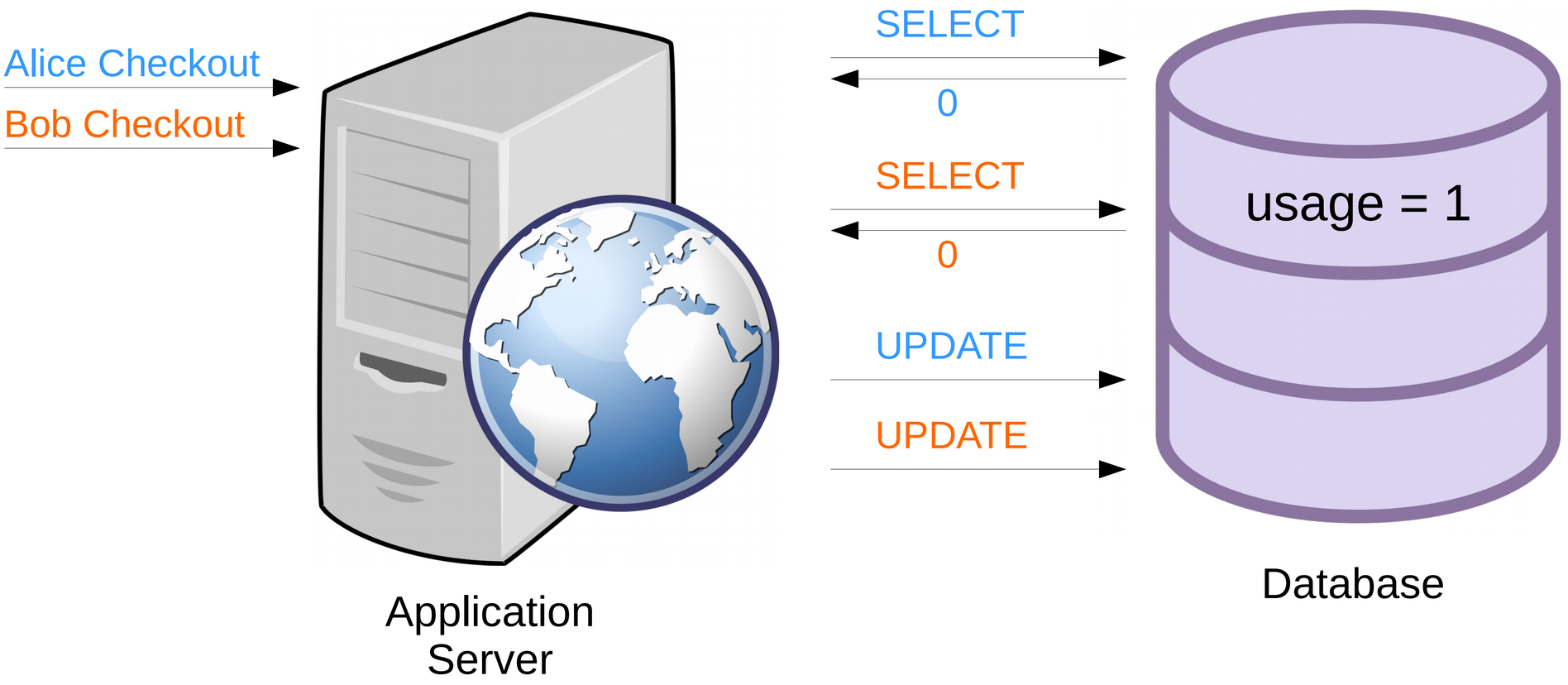
Serializability of API Requests

Serializability of Database Transactions

Non-Transactional Implementation

```
def checkVoucher(code):
    usage = readUsage(code)
    if (usage == 0):
        markUsed(code)
```

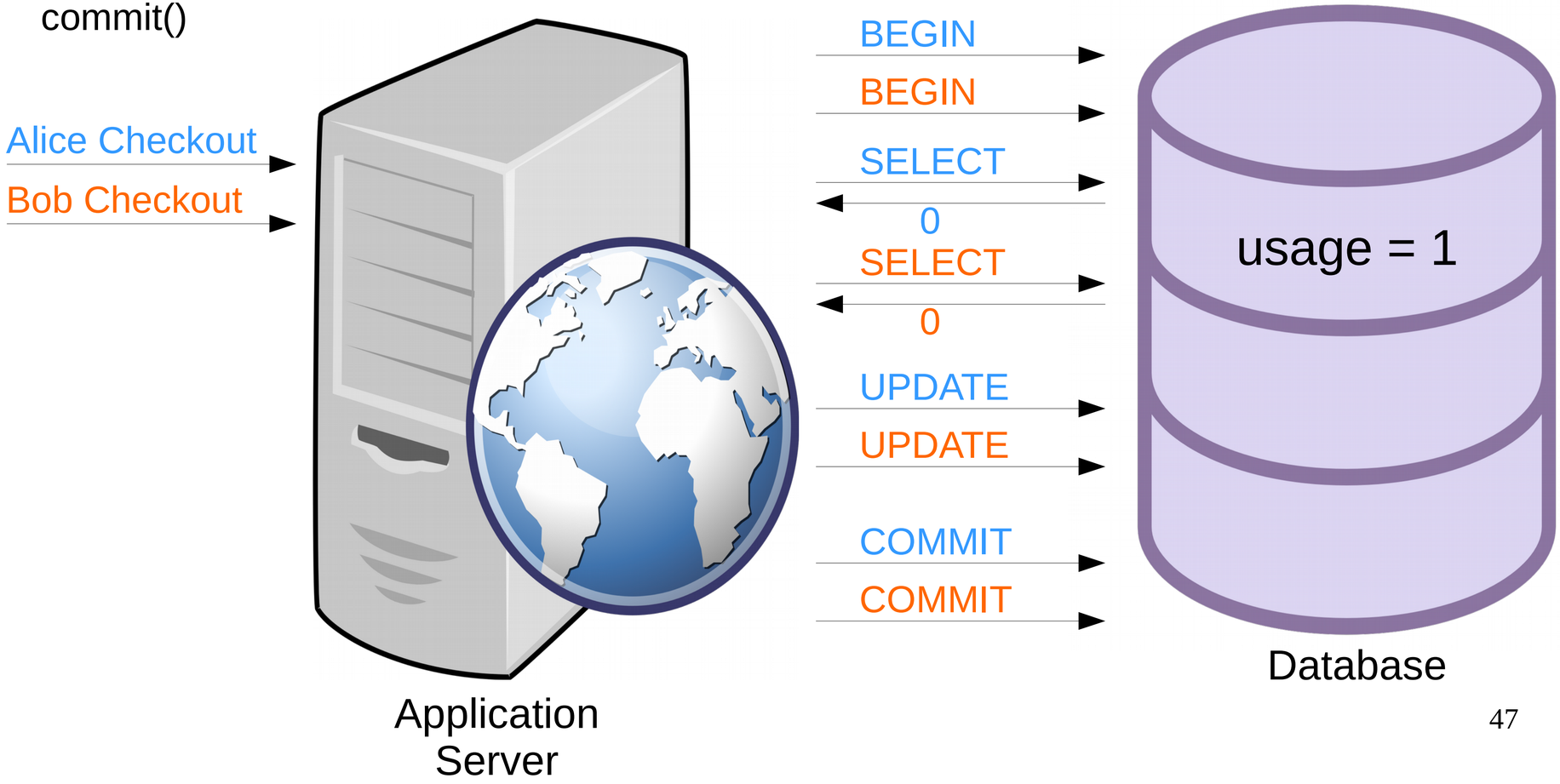
SELECT usage FROM voucher WHERE code = HNUHY
UPDATE voucher SET usage = 1 WHERE code = HNUHY



Transactional Implementation

```
def checkVoucher(code):
    beginTxn()
    usage = readUsage(code)
    if (usage == 0):
        markUsed(code)
    commit()
```

```
BEGIN TRANSACTION
SELECT usage FROM voucher WHERE code = HNUHY
UPDATE voucher SET usage = 1 WHERE code = HNUHY
COMMIT
```



Two Sources of Vulnerabilities

- Databases providing weak isolation may exhibit non-serializable behavior

```
def checkVoucher(code):
    beginTxn()
    usage = readUsage(code)
    if (usage == 0):
        markUsed(code)
    commit()
```

- Programmers may code transactions incorrectly

```
def checkVoucher(code):
    usage = readUsage(code)
    if (usage == 0):
        markUsed(code)
```

Many Databases Allow This Anomaly

Database	Default Isolation	Maximum Isolation
Actian Ingres 10.0/10S	✓	✓
Aerospike	✗	✗
Akiban Persistit	✓	✓
Clustrix CLX 4100	✓	✓
Greenplum 4.1	✗	✓
IBM DB2 10 for z/OS	✓	✓
MySQL 5.6	✗	✓
MemSQL 1b	✗	✗
MS SQL Server 2012	✗	✓
NuoDB	✓	✓
Oracle 11g	✗	✓
Oracle Berkeley DB	✓	✓
Oracle Berkeley DB JE	✓	✓
Postgres 9.2.2	✗	✓
SAP HANA	✗	✓
ScaleDB 1.02	✗	✗
VoltDB	✓	✓

✓ = prevents anomaly

✗ = exhibits anomaly

Cart Attack (Free Items)

Cart Invariant: Total charged = value of items

- Attack:

  - System calculates total from cart
  - Attacker modifies cart concurrently
  - Payment uses old cart total
  - Order uses updated cart

- This results:

  - User pays less than actual value
  - Possibility to obtain items for free

Why Transactions Are Not Enough

Transactions alone are insufficient

- Weak isolation allows anomalies
- Reads may see outdated values
- Missing locks allow interleaving

Even with transactions:

-> Non-serializable behavior possible

-> Strong isolation or locking (SELECT FOR UPDATE) required

Proposed Solution (2AD)

Abstract Anomaly Detection (2AD)

- Goal:

  - Detect potential concurrency vulnerabilities

- Approach:

  - Analyse SQL traces (logs)
  - Model database operations
  - Detect anomalies automatically

2AD Workflow

2AD Process:

1. Collect database traces
   - SQL queries from API calls

2. Build abstract history graph
   - Nodes = operations
   - Edges = conflicts

3. Detect cycles
   - Cycle = possible anomaly

Abstract History Graph

Abstract history representations

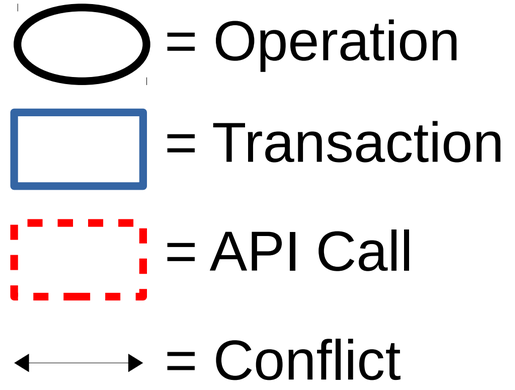
- Operations: read/write queries
- Transactions: group operations
- API calls: group transactions

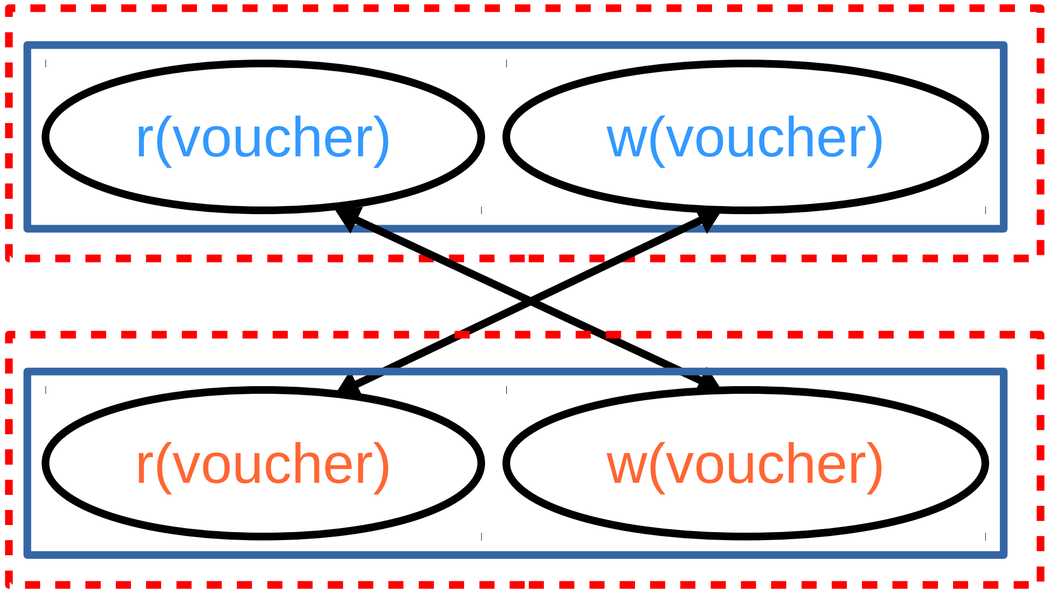
Conflicts:
- Same data accesed
- At least one is a write

Key insight:
- Cycle in graph -> non-serializable execution

Abstract History Graph

BEGIN TRANSACTION
SELECT usage FROM voucher WHERE code = HNUHY
UPDATE voucher SET usage = 1 WHERE code = HNUHY
COMMIT
BEGIN TRANSACTION
SELECT usage FROM voucher WHERE code = HNUHY
UPDATE voucher SET usage = 1 WHERE code = HNUHY
COMMIT

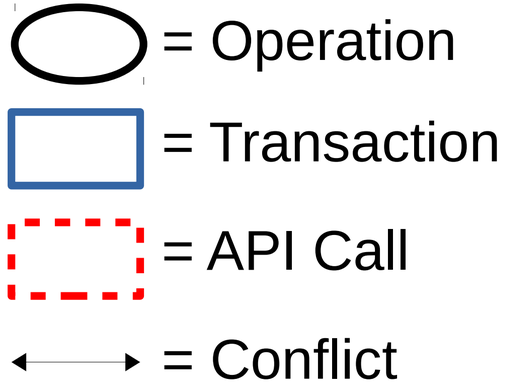


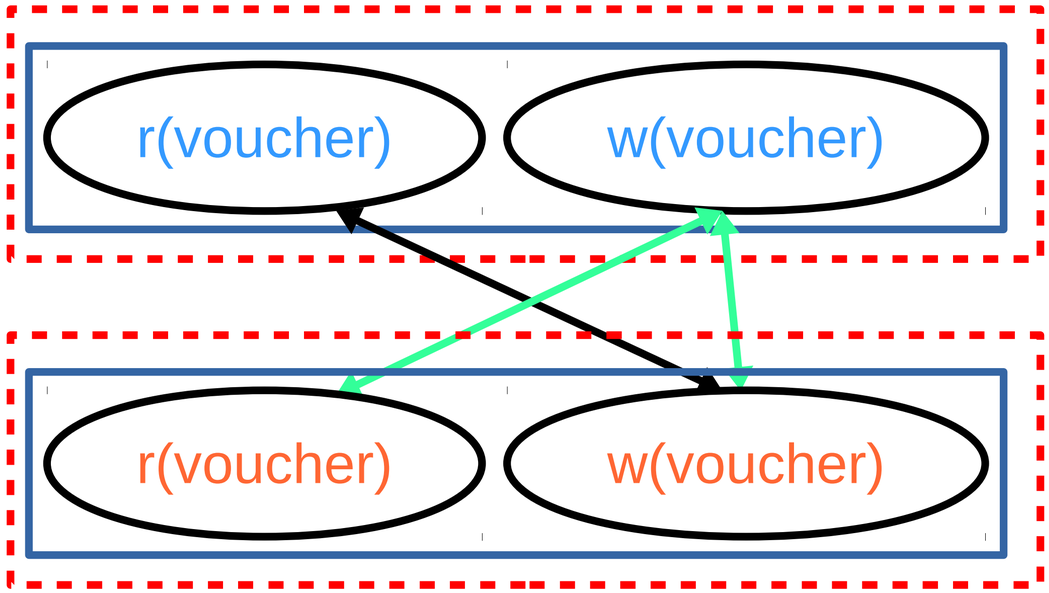


1. Add node for each operation
2. Add supernode for each transaction
3. Add super-supernode for each API call
4. Add edge for each conflict

Abstract History Graph

BEGIN TRANSACTION
SELECT usage FROM voucher WHERE code = HNUHY
UPDATE voucher SET usage = 1 WHERE code = HNUHY
COMMIT
BEGIN TRANSACTION
SELECT usage FROM voucher WHERE code = HNUHY
UPDATE voucher SET usage = 1 WHERE code = HNUHY
COMMIT

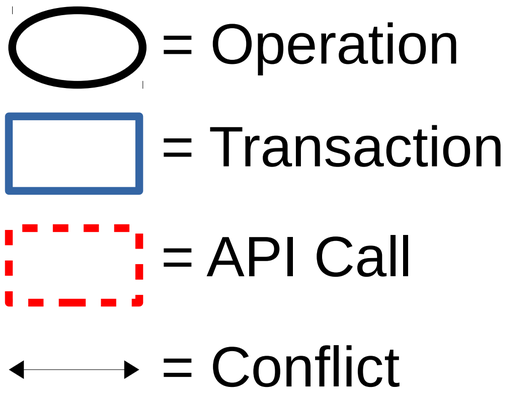


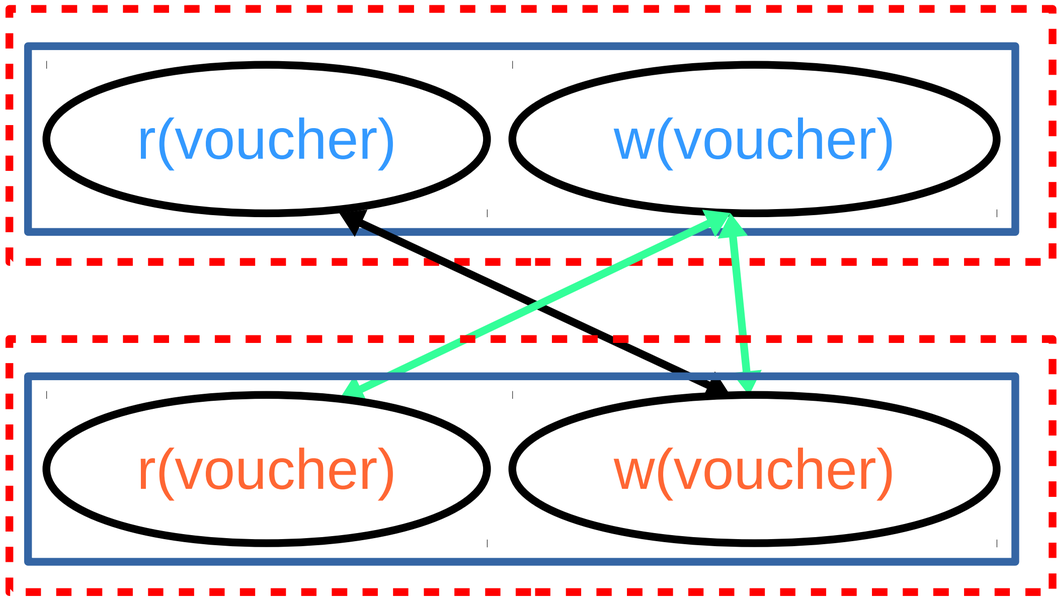


1. Add node for each operation
2. Add supernode for each transaction
3. Add super-supernode for each API call
4. Add edge for each conflict
5. Search for cycles in the graph

Abstract History Graph

BEGIN TRANSACTION
SELECT usage FROM voucher WHERE code = HNUHY
BEGIN TRANSACTION
SELECT usage FROM voucher WHERE code = HNUHY
UPDATE voucher SET usage = 1 WHERE code = HNUHY
COMMIT
UPDATE voucher SET usage = 1 WHERE code = HNUHY
COMMIT





1. Add node for each operation
2. Add supernode for each transaction
3. Add super-supernode for each API call
4. Add edge for each conflict
5. Search for cycles in the graph

Key Theoretical Insight

Theorem (Informal)

- An anomaly exists <-> There is a non-trivial cycle in the graph
- Implication:

  - No need to enumerate all executions

  - Cycle detection is sufficient

Evaluation Setup

Evaluation:

- 12 eCommerce platforms
- 4 programming languages
- Over 2 millions deployments
- Tested invariants:
  - Inventory (stock ≥ 0)
  - Voucher (usage limit)
  - Cart (correct pricing)

Results

22 vulnerabilities discovered

- Breakdown:
  - 9 inventory issues
  - 8 voucher issues
  - 5 cart issues

- Key finding:
  - 17 due to incorrect transaction usage
  - Only 5 due to weak isolation

Analysis Results

22 new vulnerabilities!

✗ = vulnerable, ✔ = not vulnerable

Application	Language	Inventory	Voucher	Cart
Opencart	PHP	✗	✗	✔
Prestashop	PHP	✗	✗	✔
Magento	PHP	✗	✗	✔
WooCommerce	PHP	✗	✗	✔
Spree	Ruby on Rails	✔	✔	✔
Ror_ecommerce	Ruby on Rails	✗	N/A	✗
Shoppe	Ruby on Rails	✗	N/A	✗
Oscar	Python (Django)	✗	✗	✔
LFS	Python (Django)	✗	✗	✗
Saleor	Python (Django)	✗	✗	N/A
Broadleaf	Java (Spring)	N/A	✗	✗
Shopizer	Java (Spring)	N/A	N/A	✗

2M+ sites at risk

4 different languages

5 errors due to DB default isolation

Analysis Results

22 new vulnerabilities!

✗ = vulnerable, ✔ = not vulnerable

Application	Language	Inventory	Voucher	Cart
Opencart	PHP	✗	✗	✔
Prestashop	PHP	✗	✗	✔
Magento	PHP	✗	✗	✔
WooCommerce	PHP	✗	✗	✔
Spree	Ruby on Rails	✔	✔	✔
Ror_ecommerce	Ruby on Rails	✗	N/A	✗
Shoppe	Ruby on Rails	✗	N/A	✗
Oscar	Python (Django)	✗	✗	✔
LFS	Python (Django)	✗	✗	✗
Saleor	Python (Django)	✗	✗	N/A
Broadleaf	Java (Spring)	N/A	✗	✗
Shopizer	Java (Spring)	N/A	N/A	✗

2M+ sites at risk

4 different languages

5 errors due to DB default isolation

17 errors due to improper transaction usage

How to Prevent These Attacks

Prevention Strategies:

- Correct transaction scoping
- Use stronger isolation levels
- Explicit locking (e.g. SELECT FOR UPDATE)
- Avoid multiple inconsistent reads
- Validate invariants multiple times

Strengths

Strenghts:

- Introduces new attack class
- Combines theory and practice
- Language-agnostic approach
- Real-world impact (millions of sites)

Limitations

Limitations:

- Depends on observed traces
- May produce false positives
- Does not capture all program logic
- Requires manual interpretation

Final Summary

Summary:

- Concurrency bugs can be exploited as attacks
- Weak isolation and bad transaction usage are key causes
- 2AD detects anomalies via graph analysis
- 22 real vulnerabilities found in practice

Thanks you!
Questions?