Fast Money, Slow Locks: Chronicles of Atomic Trespasses and Memorials of Transactional Craft

– or what is this TREM –

Otávio A Araújo Silva

otavioarj

NullByte 8 Nov. 2025

Agenda

1. Race Conditions & Semantics
   - Problem Framing
   - Ledger Semantics
   - Distributed Angle (Glossary)
2. Exploitation Playbook
   - Attacker Model & Recon
   - Concrete Exploits in Ledgers
   - Failure Signals (Telemetry)
   - Exploit Snippets
   - Strong Writes for Money
3. Case Studies
   - OpenSSH (2024) — Timing and Signals
   - Starbucks (2015) — Gift-Card Double-Spend
4. Testing & Chaos
5. This never happened
6. This is the TREM

1. Race Conditions & Semantics
   - Problem Framing
   - Ledger Semantics
   - Distributed Angle (Glossary)
2. Exploitation Playbook
3. Case Studies
4. Testing & Chaos
5. This never happened
6. This is the TREM

Race Conditions & Semantics

Problem Framing

- Latency windows in payment paths enable: double-spend, duplicate payments, stale authorizations, overdrafts.
- Parallelization (workers, retries, webhooks, autoscaling) scales both throughput *and* exploit surface.
- Monetary loss is non-linear: narrow races $\times$ burst concurrency $\Rightarrow$ large leakage.

Security View

- Attackers synchronize requests to the pre-commit gap (race orchestration).
- Hotspots: balance checks, hold creation, gift-card transfers, reversal/cancel flows.

Definitions: Logical vs Data Races

Logical Race - Our focus here

Correctness depends on the interleaving of operations on shared **application state** (DB rows, cache keys, queues). High level race conditions.

Data Race

Memory-model violation, unsynchronized read/write of same location causing undefined behavior. Low-level race conditions, e.g., virtual memory r/w locks violations

Why It Matters

Logical races survive **proper thread locks** and typically arise across processes/services/DBs.

Taxonomy of Race Conditions

Application/DB

- **TOCTOU**: stale check then effect (classic “check-then-act”).
- **Lost update**: concurrent writes overwrite each other; e.g. balance decrements race.
- **Write skew & phantoms**: constraints enforced via separate reads; concurrent inserts slip through.
- **ABA-like logical patterns**: value toggles and returns; naive guards pass.

Distributed System

- **Ordering races**: out-of-order queue/webhook deliveries observe inconsistent states.
- **Duplicate delivery races**: retries reapply effects if idempotency is weak.
- **Lease/lock races**: expired holders still write without fencing tokens.

Mobilization vs Settlement (Ledger Invariants)

Terminology

- **Mobilize (Hold/Authorization)**: reserve funds temporarily.
- **Settle (Capture/Transfer)**: finalize and post movement; release or consume hold.

Critical Invariants

- Same unit of value cannot be mobilized twice before settlement.
- Hold creation must not exceed available balance after concurrent holds.
- Settlement must relate to a single, valid, unconsumed hold.

Operational Concurrency: GC, Skew, Split Brain

Runtime/Infra

- **Garbage Collection pauses**: stop-the-world intervals that delay threads and expand timing windows.
- **Clock skew**: node clocks drift; time-based leases/expirations misfire (stale holders act as valid).
- **Split brain**: partition elects multiple leaders; conflicting writes occur concurrently.

Consistency Building Blocks

- **Isolation levels**: RC/RR/Serializable; weaker levels admit lost updates, write skew, phantoms.
- **Idempotency**: retries return same logical result; requires dedup state at write.
- **Exactly-once**: rarely achievable across networks; enforce invariants at the DB boundary.

Consistency Building Blocks

Isolation Levels: RC/RR/Serializable

- **RC (Read Committed)**: Transactions only read committed data; prevents dirty reads, allows non-repeatable reads and phantom reads, e.g. multiples insert/deletes inconsistency.
- **RR (Repeatable Read)**: Guarantees that if a transaction reads a value, subsequent reads will return the same value; prevents non-repeatable reads, still allow phantom reads.
- **Serializable**: Strongest isolation level; transactions execute as if they were run one after another; prevents all anomalies: lost updates, write skew, and phantoms.
- **Weaker levels trade-off**: Lower isolation levels= performance++. Admit anomalies: concurrent overwrites), write skew (constraint violations), and phantoms, e.g new rows appearing in range queries).

Consistency Building Blocks

Idempotency

- **Definition**: An operation is idempotent when executing it multiple times produces the same logical result as executing it once.
- **Retry safety**: Enables safe retries in distributed systems where network failures may cause duplicate requests; critical for at-least-once delivery semantics.
- **Deduplication state**: Requires maintaining state to detect and filter duplicate operations; typically implemented using request IDs, timestamps, or sequence numbers stored at write time.
- **Examples**: SET operations are naturally idempotent; INCREMENT operations require deduplication tokens to become idempotent.

Consistency Building Blocks

Exactly-Once Semantics

- **The challenge**: True exactly-once delivery is theoretically impossible across networks due to fundamental distributed systems problems (network partitions, timeouts, crashes).
- **Practical approach**: Instead of guaranteeing exactly-once at the network level, enforce invariants and consistency rules at the database boundary.
- **DB-level guarantees**: Use transactions, unique constraints, and conditional writes to ensure logical exactly-once semantics even with at-least-once delivery.
- **Implementation pattern**: Combine idempotency keys with transactional boundaries; the database becomes the source of truth for deduplication and consistency enforcement.

How Attackers Find and Time Races

Recon

- Identify multi-step flows: check → authorize/hold → capture/settle → release.
- Probe endpoints that move money or create ledger effects (transfer, reload, reversal).
- Observe retries/timeouts/webhook behavior; infer idempotency gaps.

Timing Orchestration

- Burst concurrent requests; jitter control to land between check and commit.
- Exploit server/client retries; force ambiguous failures (timeouts) to trigger replays.
- Abuse cross-service latencies (API→queue→worker→DB) to desynchronize guards.

Exploit: Double Mobilization (Two Holds, One Balance)

Mechanics

- Two workers **A** and **B** read "balance $\geq N$" before any mutation.
- Both insert holds based on the same stale snapshot; weak isolation (RC/RR) admits it.
- Synthetic liquidity emerges; settlement later drains real funds or creates debt.

Failure Conditions

- Split check/mutate across calls/services.
- No single-statement guard; no unique idempotency key tied to effect.

How Races Surface in Metrics

Quantitative Indicators

- Rising rate of **duplicate-key conflicts** (idempotency table) vs throughput.
- Ratio: holds created vs holds released/captured in short intervals.
- Inconsistent ledger deltas (balance) across replicas (local copies); spike in p99 latency[a] at transactions endpoints.

[a] Highest latency of 1% slowest response

Qualitative Indicators

- Chargeback-like symptoms without external processor issues.
- Intermittent “insufficient funds” after successful authorizations.

Anti-Pattern (Pseudo-SQL) — Split Check and Mutate

Don't: Predicate Not Bound to Effect

```
-- Workers A and B execute concurrently
bal = SELECT balance FROM accounts WHERE id = :u;
IF bal >= :n THEN
  INSERT INTO holds(user, amount) VALUES(:u, :n);
END IF;
```

Where It Fails (A & B)

- **A** and **B** both observe the same pre-state **before** any mutation.
- The check is detached from the write; interleavings admit **two holds**.
- Under RC/RR isolation, no serialization prevents the lost-update style outcome.

Atomic Predicate + Effect (SQL Pattern)

Do: Bind Predicate to Effect

```
BEGIN;
UPDATE accounts
SET balance = balance - :n
WHERE id = :u AND balance >= :n;
-- rowcount == 1 -> success; else -> insufficient funds
INSERT INTO holds(user, amount, idempotency_key)
VALUES(:u, :n, :k);
COMMIT;
```

Why This Works

- Single-statement guard; success gated by affected row count: **no separate read**.
- Effect coupled to a unique **idempotency key**; replays = same result.

Idempotency & Async Events

Transactional Idempotency

- Unique key per logical funds request; persisted with the effect **in the same transaction**.
- On replay: return stored result; never reapply state changes.

Treat Async Events as Participants

- Webhooks/timeouts fenced/serialized with the write (version checks, fencing tokens).
- Reject mid-flight state flips unless tied to the same commit boundary.

1. Race Conditions & Semantics

2. Exploitation Playbook

3. Case Studies
   - OpenSSH (2024) — Timing and Signals
   - Starbucks (2015) — Gift-Card Double-Spend

4. Testing & Chaos

5. This never happened

6. This is the TREM

OpenSSH (2024): Signal-Handler Race — CVE-2024-6387 "regreSSHion"

Essence

- Timing-sensitive interaction between signal delivery and handler logic in code: LoginGraceTime(handler)$\rightarrow$async-signal-safe func(ex.,syslog)$\rightarrow$malloc/free = RCE
- Asynchronous events flip flags mid-operation; assumptions diverge pre/post handler.
- Analogy: expirations/timeouts/webhooks behave like signals; can preempt money writes.

Portable Lessons

- Treat async events as preemption points; design critical sections robust to interruption.
- Avoid "flag-based" correctness outside the transaction boundary.

Starbucks Gift-Card Double-Spend (2015): Mechanics

Observed Behavior

- Concurrent **reload/transfer** calls allowed duplicate credit between gift cards.
- Guard and effect split across services (app $\rightarrow$ API $\rightarrow$ DB) admitted interleavings.
- Idempotency not enforced end-to-end; replays landed as new effects.

Attacker Steps (Reconstructed)

1. Pick two cards: source (A) and destination (B). Ensure A passes the guard.
2. Fire k parallel transfers/reloads A$\rightarrow$B with identical parameters.
3. Induce retries via client timeouts or network jitter to widen the replay surface.
4. Observe B credited multiple times while A debits are inconsistent/singular.

Starbucks (2015): Root Causes & Exploit Envelope

Root Causes

- Non-atomic *check* (balance on A) and *mutate* (debit A, credit B).
- Weak/implicit idempotency (no unique key or dedup record tied to effect).
- Missing single-writer invariant for the pair (A,B) during transfer.

Exploit Envelope

- Window width: sub-10 ms sufficient under burst; amplified by autoscaling workers.
- Reliable under client-induced retries and partial timeouts.
- Detector: transient mismatches of #transfers vs net ledger delta per minute.

Temporal Fuzzing & Chaos for Money Paths

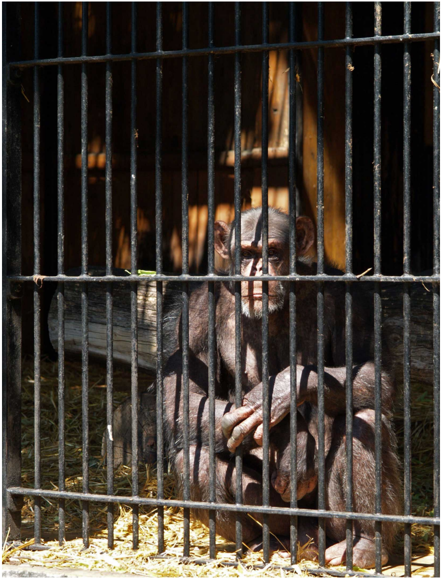
Technique

- Inject jitter between guard and effect; N workers hammer the same logical request.
- Expectation: zero duplicate holds/captures; any non-zero indicates a race.
- Metrics: dedup hit rate, conflict retries, holds vs releases/captures ratio.

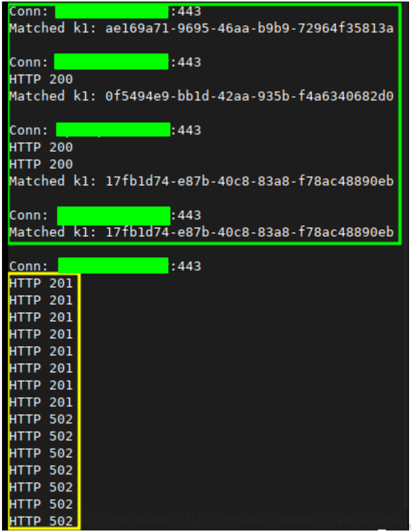
Chaos Scenarios

- Kill a replica mid-transaction; drop/reorder webhooks; expire sessions mid-write.
- Force client retries/timeouts; verify invariants under maximal interleaving.

This never happened?



This is The TREM



```
Conn: [redacted]:443
Matched k1: ae169a71-9695-46aa-b9b9-72964f35813a

Conn: [redacted]:443
HTTP 200
Matched k1: 0f5494e9-bb1d-42aa-935b-f4a6340682d0

Conn: [redacted]:443
HTTP 200
HTTP 200
Matched k1: 17fb1d74-e87b-40c8-83a8-f78ac48890eb

Conn: [redacted]:443
Matched k1: 17fb1d74-e87b-40c8-83a8-f78ac48890eb

Conn: [redacted]:443
HTTP 201
HTTP 201
HTTP 201
HTTP 201
HTTP 201
HTTP 201
HTTP 201
HTTP 201
HTTP 502
HTTP 502
HTTP 502
HTTP 502
HTTP 502
HTTP 502
HTTP 502
```

TREM - Transactional Racing Executor Monkey

```
Transactional Racing Executor Monkey v0.8 Pu

  -d int
        Delay ms between reqs.
  -h string
        Host:port override; default is addr from Host HTTP Header.
  -k    Keep-alive connections; persist TLS tunnels for every request, including while looping (-xt N).
  -l string
        Comma-separated request RAW HTTP/1.1 files.
  -mode string
        Mode: sync or async. (default "async")
  -o    Save *last* (per-thread) HTTP response as out_<timestamp>_t<threadID>.txt
  -px string
        HTTP proxy; http://ip:port
  -re string
        Regex (Golang) definitions file. Each line applies to a request file, respectively.
        Examples:
          One Regex line for each request file, e.g., line 1 will use regexes for request 1
          Format: regex1':key1$$regex2':key2$$...regex':keyN supports multiples regex per line/request
          Note: Use backtick (`) character, not (').
  -th int
        Threads count. (default 1)
  -u string
        Universal replace key=val every request; e.g., !treco!=Val replaces !treco! to Val in every request, multiple times if matched.
  -x int
        When looping, chain request x to N, where x is -l [1..x..N], default disabled.
         Ex: -l "req1,req2,req4" -x 2, does reqs 1 to 4, then iterates -xt times from req2 to req4 (default -1)
  -xt int
        Requests loop count:
         0=infinite
        -1= zero loops
```

Questions?


Jack the baboon worked on the railway system in South Africa for
9 years without ever making a single mistake.

This is TREM

otavioarj/TREM