The Read-Write Gap: Web Race Conditions Beyond TOCTOU

April 26, 2026

In October 2023, James Kettle published a technique that made remote web race conditions as reliable as local ones: the single-packet attack [Kettle, 2023 (https://portswigger.net/research/the-single-packet-attack-making-remote-race-conditions-local)]. By batching the tail frames of many HTTP/2 requests into one TCP segment, an attacker can eliminate network jitter and force a server to process dozens of requests inside a window of roughly one millisecond [Kettle, 2023 (https://portswigger.net/research/the-single-packet-attack-making-remote-race-conditions-local)].

That change matters because the classical threat model for race conditions (single-process filesystem TOCTOU) does not describe what most web applications are actually vulnerable to. Your attacker is not swapping an inode between `stat()` and `open()`. The attacker is not violating HTTP/2. Every frame they send is spec-compliant. The vulnerability lives entirely in your handler's assumption that concurrent requests arrive serialized in time.

This post defines the class precisely, walks through the four patterns that make it observable, and connects the single-packet attack back to the application-layer gaps it exploits. By the end, you should be able to open one of your own handlers and name the read-write gap it ships.

The Vulnerability, Precisely

CWE-362 defines a race condition as a code sequence that requires temporary, exclusive access to a shared resource, but where a timing window allows another concurrent code sequence to modify that resource [MITRE, n.d. (https://cwe.mitre.org/data/definitions/362.html)]. The definition is intentionally protocol agnostic. In a web-application, the specifics sharpen considerably:

- **The shared resource** is almost always a database row, a cache entry, a session record, or an external-service record reachable by key.
- **The concurrent code sequences** are HTTP requests, typically from the same user (an attacker), often over the same connection.
- **The exclusivity requirement** is implicit in business logic, e.g. "this coupon has one use left," "this user's role is X", and the application fails to enforce it at the storage layer.
- **The timing window** opens when a handler reads state into application memory, runs logic, and writes state back. Between read and write, any invariant the read observed can be violated by a concurrent writer.

HTTP's stateless-request model is not the root cause, but it shapes the attack surface. The root cause actually stems from the deployment topology layered on top of it: short-lived request handlers distributed across many processes or instances, with no shared in-memory state. An in-process mutex keyed by user is simple enough to build inside a single process, but in a horizontally scaled deployment it governs nothing[1]. Request A for user U may land on one instance, request B on another, and the mutex on the first instance is invisible to the second. The only coordination primitive that spans all concurrent handlers is the datastore. When the datastore is treated as a simple key-value store instead of the system of record for concurrency control, the read-write gap becomes exploitable.

The simplest possible case, as a trace:

```
TICK  REQ A                          REQ B                          COUPON
ROW
----  -----------------------------  -----------------------------  ---------
----------
 t0   SELECT uses_remaining           -
uses_remaining = 1
 t1   reads 1, passes "> 0" check     SELECT uses_remaining
uses_remaining = 1
 t2   apply_discount(A)               reads 1, passes "> 0" check
uses_remaining = 1
 t3   UPDATE uses_remaining = 0       apply_discount(B)
uses_remaining = 0
 t4   -                               UPDATE uses_remaining = 0
uses_remaining = 0
```

Both requests pass the check. Both apply the discount. The row is correctly decremented once, because the second `UPDATE` produces the same result, but the business invariant is violated. No single line of code is wrong in isolation. The composition is wrong.

Why TOCTOU Does Not Capture It

CWE-367 is defined broadly: any check-then-use pattern on a shared resource. A recent CVE filed under CWE-367 is Grafana CVE-2026-21725, where a deleted-then-recreated data source could be re-deleted without permission inside a roughly thirty-second window [NIST NVD, 2026 (https://nvd.nist.gov/vuln/detail/CVE-2026-21725)]. In practice, though, the canonical TOCTOU framing is still heavily filesystem-shaped: a program checks a property of a named resource, such as existence, ownership, or permissions, then acts on that same resource under the assumption the property still holds. That model fits one of four web-app patterns below and describes the other three poorly or not at all: they involve multiple writers, multiple resources, or invariants that span sequences of writes rather than the state of a single object at any given moment. "Time of check, time of use" is the wrong unit of analysis for any of them, even when the CWE technically fits.

The distinction that matters is threefold.

1. **Scope of resources**. Classical TOCTOU is one check, one resource: the check and the use are about the same named object. Web-app races span broader shapes: one endpoint racing itself over one row (counter overrun), two endpoints racing over one row (state-machine collapse), one endpoint racing itself across two resources where only one is transactional (idempotency), and the middleware-versus-storage races where the "resource" is a cache copy that disagrees with what the authoritative store has to say (stale attributes).
2. **Participants**. Classical TOCTOU is two processes racing on one host. Web-app races are N concurrent requests landing within milliseconds across M application instances, each holding a share of the state and all pointed at one datastore. The scaling dimension is not "attacker process versus victim process" but "how many parallel handlers can I get the server to interleave inside a single read-write gap."
3. **Invariant locus.** Classical TOCTOU asks whether a property observed at time T1 still holds at time T2 for the same resource. Web-app invariants are business rules ("one discount per coupon," "one charge per idempotency key," "the identity that receives the reset confirmation owned the account when the token was issued") that are properties of the *sequence of writes*, not of any resource's state at any given moment. Framing these as "time of check, time of use" is dishonest: the invariant must be expressed as a constraint the storage layer enforces on every commit, not as an assertion the application makes between two reads.

Treating web races through the classical TOCTOU lens is not wrong, but it leaves three of the four patterns below invisible. The useful generalization is not "time of check, time of use." It's the **read-write gap**: any interval between observing state and committing a mutation that depends on that observation[2].

A Formal Pattern Taxonomy

The patterns sort cleanly along two axes plus one orthogonal case:

- **Axis 1, endpoint count**: does the race involve concurrent requests to a single endpoint, or coordination across multiple endpoints?
- **Axis 2, resource count**: does the race involve a single shared resource, or multiple resources where the invariant spans between them?

That gives a 2x2 matrix. The orthogonal case is cross-request cached state, which is an in-memory-versus-persistent boundary rather than a write-ordering problem, but it belongs in the same vein, since attackers still exploit it with the same primitives.

Pattern 1: Single-Endpoint / Single-Resource

Shape. N concurrent requests to the same endpoint, all reading and writing the same row. The endpoint's handler is logically correct for one request at a time.

Example. Coupon or quota overrun. Realized in the wild as CVE-2024-45300 (alf.io (http://alf.io)), where the promo code limit check and the usage recording were separated by a window an attacker could fill with parallel requests, enabling unlimited reuse of a single-use coupon [NIST NVD, 2024 (https://nvd.nist.gov/vuln/detail/cve-2024-45300)].

Trace. As shown in "The Vulnerability, Precisely" above, each request reads `uses_remaining = 1`, passes the check, applies the discount, and writes `uses_remaining = 0`. Twenty concurrent requests all see the pre-state and all proceed.

Invariant violated. `count(applications of this coupon) <= max_uses`.

Why it exists. The handler uses the datastore as a read-then-write KV store. The check is in application memory. The datastore never sees a statement that expresses the invariant.

Pattern 2: Multi-Endpoint / Single-Resource

Shape. Two different endpoints mutate the same resource, and the business invariant requires them to serialize in a particular order, but the application does not enforce that serialization.

Example. Password reset flows that separate "issue reset token" from "consume reset token," *without* atomically binding the token to the email that existed at issue time. CVE-2026-32943 (Parse Server) is a recent instance: the reset token could be consumed by multiple concurrent requests inside a short window, so an attacker who intercepted a token could race the legitimate user's reset and both requests would succeed, with the attacker's password winning [NIST NVD, 2026 (https://nvd.nist.gov/vuln/detail/CVE-2026-32943)]. The fix atomically validates and consumes the token as part of the password update flow.

Trace.

```
TICK  REQ A (reset confirm)                    REQ B (change email)
USER ROW
----  ---------------------------------------  ----------------------------
----  -------------------------
 t0   SELECT email, token WHERE id=U            -
email=victim, token=X
 t1   reads victim, X; validates token          UPDATE email='attacker' WHERE
id=U  email=attacker, token=X
 t2   UPDATE password=hash(new) WHERE id=U      -
email=attacker, password=new
 t3   send reset confirmation to 'victim'       -
(confirmation mail lost)
```

Neither endpoint on its own yields a violation of the invariant. The reset endpoint validates the token. The email endpoint validates ownership. The race is across the two, and the resource (the user row) is the shared thing.

Invariant violated. “The identity that receives the reset confirmation is the identity that owned the account when the token was issued.”

Why it exists. Multi-step business flows are written as independent endpoints with independent validation. The state machine exists on paper; it is **not** enforced by the datastore.

Pattern 3: Single-Endpoint / Multi-Resource

Shape. One endpoint writes to multiple resources, and the invariant that makes the write safe spans all of them. Idempotency-key handling is the archetypal case.

Example. A payments endpoint that checks an idempotency table for a matching key, and if none exists, charges the card and then inserts the key. Brandur’s reference implementation of Stripe-like keys explains the correct pattern: the insert must be the gate, not post-hoc bookkeeping, because a `UNIQUE` constraint is the only primitive that guarantees exactly one request wins [Brandur, n.d. (https://brandur.org/idempotency-keys)]. Stripe’s own API docs make the same point [Stripe, n.d. (https://docs.stripe.com/api/idempotent_requests)].

Trace.

```
TICK  REQ A                                        REQ B
IDEM TABLE   CARD
----  -------------------------------------------  ------------------------------
---------  -----------  --------------
 t0   SELECT * FROM idem WHERE key=K             -
(empty)      -
 t1   returns empty; proceed                     SELECT * FROM idem WHERE key=K
(empty)      -
 t2   stripe.charge(...)                         returns empty; proceed
(empty)      charged
 t3   INSERT idem (key=K, response=...)          stripe.charge(...)
key=K        charged twice
 t4   return 200                                 INSERT idem (key=K, ...) ->
conflict     key=K        charged twice
```

Two resources: the idempotency table and the external card ledger. The invariant (“at most one charge per key”) requires atomic coordination between them. The idempotency table check in application memory cannot provide it.

Invariant violated. “At most one side-effect per idempotency key.”

Why it exists. Idempotency is treated as a cache-and-replay concern (“have I seen this key?”) rather than a concurrency-gating concern (“can I claim this key?”).

Pattern 4: Cross-Request Cached State

Shape. Application middleware resolves user attributes (role, plan, quota) at request entry, caches them on the request context, and every downstream check uses the cached value. If the attribute changes mid-request or mid-flow, the handler continues to make decisions against stale data.

Example. An admin revokes a user's elevated role (say, demoting a departing employee from `admin` to `member` on tenant `T`) while that user has an in-flight privileged request whose authorization is gated by a middleware-cached role. The user fires the privileged request so that its middleware resolves before the revocation commits. Middleware caches `role=admin`, the revocation commits, and the privileged handler reads the cache and executes an action the live database would now forbid. The same shape applies to plan downgrades, quota resets, feature-flag toggles, and trial expirations.

Trace.

```
TICK  REQ A (privileged action, by U)            REQ B (admin revokes U's role)
MEMBERSHIPS       ROLE CACHE (Req A)
----  -----------------------------------------  ------------------------------
----------  ---------------  ------------------
 t0   middleware: resolve role for user U        -
U=admin on T      U=admin
 t1   -                                          UPDATE role='member' WHERE
user=U         U=member on T     U=admin (stale)
 t2   handler: check role cache (=admin)         return 200
U=member on T     U=admin (stale)
 t3   handler: execute admin-only action         -
U=member on T     U=admin (stale)
```

Request A's middleware snapshot predates the revocation and Request A's handler trusts that snapshot long after it has stopped matching the current state.

Invariant violated. "Authorization decisions reflect the user's current attributes, not attributes at request entry."

Why it exists. Request-scoped caching is a performance optimization that implicitly assumes request duration is negligible. Under a single-packet attack with N parallel requests, that assumption fails.

Axis note. This pattern does not fit the endpoint/resource matrix cleanly. It is a persistence-boundary race: the cached copy and the authoritative store disagree for the duration of the request[3]. It belongs in the taxonomy because attackers exploit it with the same tooling, but the underlying gap is in middleware, not storage.

The Single-Packet Attack as Exploitation Enabler

The four patterns above predate HTTP/2. Remote exploitation of the wider read-write gaps was already practical over HTTP/1.1 via last-byte sync, with a spread of ~ four milliseconds. What changed in 2023 is that the narrower gaps became reliably exploitable from anywhere on the internet.

The technique exploits HTTP/2's stream multiplexing. The attacker:

1. Opens one TCP connection, negotiates HTTP/2.
2. For each of N requests, sends the `HEADERS` frame and all but the last byte of the `DATA` frame. These frames arrive at the server, but none of the streams are "complete" because the finalizing data byte has not been sent.
3. Waits for the server to buffer and settle
4. Sends a `PING` frame to prime the network path (the OS stops buffering the first frame after a delay)[4].
5. Packs the final `DATA` frames for all N streams into a single TCP segment and writes them back to back.

The server receives the final bytes effectively simultaneously. All N requests become eligible for processing within roughly a millisecond of each other. Kettle's benchmarks show a median spread of about one millisecond across 20 concurrent requests, with a standard deviation under half a millisecond. Last-byte sync, the HTTP/1.1 analogue, achieves a median spread of about four milliseconds: roughly four times worse, but still sufficient for a majority of web race windows[5].

The technique inherits from Van Goethem et al. (2020) (https://www.usenix.org/conference/usenixsecurity20/presentation/van-goethem), *Timeless Timing Attacks*, which established that concurrency-based timing observations over a multiplexed protocol are network-jitter-independent. The single-packet attack applies the same primitive to race exploitation rather than timing side channels.

RyotaK's 2024 first-sequence-sync extension (https://flatt.tech/research/posts/beyond-the-limit-expanding-single-packet-race-condition-with-first-sequence-sync/) coordinates TCP sequence numbers and IP fragmentation across multiple packets, breaking the ~65,535-byte ceiling that capped the original technique at roughly 20-30 streams on a standard 1500-byte MTU. The extended version scales to dozens of streams without losing synchronization.

What this means for the patterns above. The server-side read-write gap in each pattern is typically 100 microseconds to 10 milliseconds wide: a database round-trip plus whatever application logic sits between read and write. The single-packet attack lands all attacker requests inside that window with high reliability. Network-layer defenses see a single packet and cannot distinguish twenty concurrent attacks from one well-formed request. The only durable defense boundary is one the application enforces at the storage layer.

Defense Primitives

Each pattern has an underlying primitive that closes the gap. Full implementation will be covered in a companion post later down the line; what follows is enough to know the correct direction per pattern.

Counter Overruns

Atomic `UPDATE` with the invariant in the `WHERE` clause. A zero-row result means the coupon was already exhausted. The check and the decrement are one statement. Postgres (and many other relational DBs) serialize concurrent updates to the same row at the engine level under any isolation[6]. There is no application-visible window.

Multi-endpoint State Machines

Re-validate prerequisites atomically with the mutation. The Parse Server fix for CVE-2026-32943 is the canonical shape: token validation and password update happen in one statement or one transaction that takes the relevant row lock. Do not trust values the caller provided or that middleware resolved earlier. Every mutation re-reads its prerequisites inside the same transaction that performs the mutation.

Idempotency

Insert the key first, under a unique constraint. If the insert returns zero rows, another request owns this key. The primary key constraint is the gate, not a post-hoc lookup. Stripe's pattern uses explicit state (`pending`, `succeeded`, `failed`) with a response cache keyed to the final state.

Stale Cached State

Re-resolve inside the mutation's transaction. Middleware-resolved attributes are fine for coarse-grained routing and logging. For any decision that gates a mutation, read the attribute inside the same transaction that performs the mutation and lock the relevant row. Accept the contention cost.

What Does Not Work

Application-level mutexes (process-local; effective only when routing pins all requests for the contested key to one instance, e.g. sticky sessions, actor systems, or single-leader writes, which is not typical stateless web deployment). Redis locks without fencing tokens (documented failure modes). Rate limiting (one TCP segment counts as one arriving unit). SERIALIZABLE isolation as a blanket fix: it can close some database races, but it does not replace explicit constraints, atomic claims, or side-effect ordering.[7].

Key Takeaways

- TOCTOU is the narrower instance of a broader class. The web-app generalization is the **read-write gap**: any interval between observing state and writing a mutation dependent on that observation.
- Four patterns cover the observable cases: single-endpoint/single-resource, multi-endpoint/single-resource, single-endpoint/multi-resource, and cross-request cached state. The first three sit on a 2x2; the fourth is orthogonal but exploited with the same primitives.
- The single-packet attack eliminated network jitter as a defense. Application-layer atomicity at the storage boundary is the only durable mitigation.
- Each pattern has one root-cause-aligned defense that closes the gap at the storage layer: atomic `UPDATE` with invariant in `WHERE`, transactional re-validation, unique-constraint-gated idempotency, and in-transaction attribute re-resolution. Full implementation is a separate concern.

References

Kettle, J. (2023). *The Single-Packet Attack: Making Remote Race-Conditions 'Local'.* PortSwigger Research. https://portswigger.net/research/the-single-packet-attack-making-remote-race-conditions-local

Kettle, J. (2023). *Smashing the State Machine: The True Potential of Web Race Conditions.* PortSwigger Research. https://portswigger.net/research/smashing-the-state-machine

MITRE. *CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization*. https://cwe.mitre.org/data/definitions/362.html

MITRE. *CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition.* https://cwe.mitre.org/data/definitions/367.html

NIST NVD. *CVE-2024-45300: alf.io promo code race condition.* https://nvd.nist.gov/vuln/detail/cve-2024-45300

OWASP. *Top 10 for Business Logic Abuse, BLA1:2025 Action Limit Overrun.* https://owasp.org/www-project-top-10-for-business-logic-abuse/docs/the-top-10/action-limit-overrun

NIST NVD. *CVE-2026-32943: Parse Server password reset token race condition.* https://nvd.nist.gov/vuln/detail/CVE-2026-32943

NIST NVD. *CVE-2026-21725: Grafana TOCTOU on data source delete-then-recreate.* https://nvd.nist.gov/vuln/detail/CVE-2026-21725

Brandur. *Implementing Stripe-like Idempotency Keys in Postgres.* https://brandur.org/idempotency-keys

Stripe. *Idempotent Requests.* Stripe API Reference. https://docs.stripe.com/api/idempotent_requests

PortSwigger. *Web Security Academy: Race Conditions.* https://portswigger.net/web-security/race-conditions

Van Goethem, T., Pöpper, C., Joosen, W., Vanhoef, M. (2020). *Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections.* USENIX Security 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/van-goethem

RyotaK (2024). *Beyond the Limit: Expanding Single-Packet Race Condition with First-Sequence Sync.* Flatt Security Research. https://flatt.tech/research/posts/beyond-the-limit-expanding-single-packet-race-condition-with-first-sequence-sync/

PostgreSQL Global Development Group. *PostgreSQL Documentation: Explicit Locking.* https://www.postgresql.org/docs/current/explicit-locking.html

Footnotes

1. This assumes the typical stateless-handler topology where any request can land on any instance. Sticky sessions, actor systems, or consistent-hash routing that pin all requests for a given key to one instance can make a process-local mutex sufficient, but couples correctness to the routing layer. ↩
2. "Read-write gap" is a framing convenience, not a formal taxonomy term. It overlaps with the sub-state race windows Kettle catalogs in *Smashing the State Machine*, but emphasizes the storage-layer gap rather than the state-machine perspective. ↩
3. Pattern 4 *can* be reframed as a special case of Pattern 2 if you treat the middleware cache and the authoritative row as two separate resources. The taxonomy keeps them distinct because the defense surface differs: Pattern 2 fixes live in endpoint handlers, whereas Pattern 4 fixes live in the middleware design. ↩

4. This step glosses over the kernel-level mechanics. The relevant behaviors are Nagle's algorithm and TCP send coalescing, both of which the attacker tooling typically disables (via `TCP_NODELAY`) or works around with explicit timing to keep the final segment intact. ↩
5. These numbers come from Kettle's Melbourne-to-Dublin benchmark and depend on route, server load, and TLS overhead. The absolute spread varies in practice; the durable result is the relative improvement over last-byte sync. ↩
6. This applies specifically to write-write conflicts on the same row: an `UPDATE` takes a row-level exclusive lock and serializes concurrent updaters. It does not extend to phantom-style read-write or write-skew anomalies, which require stricter isolation. ↩
7. `SERIALIZABLE` only protects state the database can observe inside the transaction. It does not order external side effects, and on hot paths it often means retries, aborts, or contention. ↩

---

https://blog.gtfo.dev/blog/race-conditions-read-write-gap/