Technical Fairness Analysis

Pick a Winner -- Contest Winner Selection Process

Prepared for: Legal defense against allegations of fraudulent draw

Date: 2026-05-27 (v2 -- supersedes v1 dated 2026-03-28)

Complete forensic analysis of the winner selection pipeline, data integrity safeguards, cryptographic audit trail, reproducible randomization methodology, and the public verifiability surface added in the 2026-05-27 hardening

CONFIDENTIAL

1. Executive Summary

The Pick a Winner application implements a winner selection process whose results are independently re-runnable by any third party and whose history is cryptographically anchored to the Bitcoin blockchain. For every contest drawn on or after 2026-05-27 the system publishes:

- A 256-bit cryptographically-random seed
- The SHA-256 hash of the sorted list of eligible commenter IDs
- The downloadable list of those IDs
- A hash-chained audit log of every administrative action on the contest
- An OpenTimestamps proof of the audit-chain head that, once Bitcoin confirms it (~6 hours after the event), cannot be altered without rewriting the Bitcoin blockchain

Anyone can plug these into a five-line Ruby snippet and recompute the exact winners; anyone can run the standard ots verify CLI against the proof bytes and confirm the audit history is anchored to a specific Bitcoin block. These two independent-reproduction properties together are the central fairness claim. The rest of this document describes the structural protections that make them hold.

Put plainly: it's a "we can't backdate or rewrite contest results without you noticing" guarantee. The system does not, and cannot, prevent a sufficiently privileged database administrator from issuing destructive SQL. What it does ensure is that any such rewrite leaves the audit chain, the published JSON snapshots, and the Bitcoin-anchored proof in a state that no longer agrees -- a mismatch any external party can spot with ots verify and a previously-saved snapshot. Fairness here is enforced by detectability, not by trust in the operator.

What this document claims:

- Reproducible random selection -- every draw records its seed, the canonical sorted ID list, and a SHA-256 hash of the input; anyone can re-run it. See §3.
- Database-level winner immutability -- once a contest reaches drawn status, the columns that hold winner / disqualification state cannot be modified by application code, raw SQL, or anything short of a PostgreSQL superuser disabling the trigger. See §7.
- Append-only, hash-chained audit log -- every administrative action is cryptographically linked to its predecessor; any deletion or rewrite of an audit row breaks the chain detectably. See §8.
- Daily automated chain verification -- a background job verifies the chain for recently-active contests and raises a critical SystemAlert on any break. See §8.4.
- Bitcoin-anchored timestamps -- every contest's audit-chain head is committed hourly to OpenTimestamps and confirmed in a Bitcoin block within ~6 hours. The proof is downloadable. Once anchored, the chain head cannot be moved without rewriting the Bitcoin blockchain. See §9.7.
- Read-only public audit-log API -- anyone can poll, snapshot, and diff the full event history of any contest. The endpoint is mounted at two equivalent paths (token-keyed and slug-keyed) so a reader who arrives via either of the two public results URLs can reach the audit log by appending a single path segment. Tampering becomes detectable to any external watchdog the moment it polls. See §9.6.
- Step-by-step verification guide -- a dedicated page at /verify-fairness walks any reader through six progressive levels of verification, with the trust threshold for each level stated explicitly.
- Per-contest verification helper -- /results/<token>/verify gathers every URL needed to check one specific contest into one page. See §14.
- Full participant transparency -- winners, disqualified entries, the entire eligible-IDs file, the seed, and the audit chain are publicly accessible from every contest's results page. Each contest is reachable at two equivalent public URLs: /facebook_page_contests/<slug> (the SEO-friendly URL the operator typically shares) and /results/<token>/<slug> (the token-keyed canonical URL). Both render the verification badge inline; the seed/hash/downloads/snippet live on the per-contest verify helper at /results/<token>/verify, one click away from either entry point.

What this document does NOT claim:

- Contests drawn before 2026-05-27 do not carry a recorded seed and are not retroactively verifiable. The public results page surfaces this honestly with a "Pre-hardening draw" notice. See §12.
- An audit event newer than its anchor's Bitcoin confirmation (~6 hours) is still vulnerable to a determined PostgreSQL superuser. Once Bitcoin confirms the anchor, that vulnerability closes. See §12.
- Source data integrity -- the Graph API response that supplied the comment list is not cryptographically signed by Facebook or Instagram. The system can prove what it drew from, but cannot prove that input matched what was publicly posted. See §12.

Where to actually check a contest: see §14 (Public Verification Surfaces) for the complete URL catalogue, and /verify-fairness for the practical step-by-step. The per-contest helper at /results/<token>/verify is the easiest entry point if you have a specific contest in mind.

2. The Selection Pipeline

Winner selection follows an 8-step sequential pipeline. Each step must complete successfully before the next begins. Any failure halts the entire process.

Step	Name	Purpose
1	ValidateToken	Verify Facebook/Instagram API access is valid
2	ResolvePlatform	Determine if Facebook or Instagram; parse post URL
3	FetchComments	Retrieve all comments from the social media post via official API
4	EnforceMinimumCommenters	Require at least 100 unique commenters for paid contests
5	EnforceEntryLimit	Check comment count against user's tier limit
6	EnforceContestFee	Calculate and require payment if applicable
7	FilterEntries	Apply eligibility rules to all entries
8	SelectWinners	Seeded shuffle from the eligible pool

Key property: The pipeline is triggered either by the contest organizer clicking "Draw Winners" or by the AutoDrawContestsJob scheduled sweep when a contest reaches its configured ends_at or scheduled_draw_at. From that point forward the process is fully automated: the organizer cannot skip steps, modify the eligible pool, or influence the random selection.

3. Random Selection Algorithm

3.1 Seeded, Reproducible Shuffle

Every draw is reproducible from a recorded seed. The system uses a single deterministic seeded-shuffle path (this supersedes the v1 dual-path Array#sample / ORDER BY RANDOM() approach, which was statistically uniform but produced non-reproducible results):

```
seed_hex = SecureRandom.hex(32)            # 256 bits from the OS CSPRNG
rng = Random.new(seed_hex.to_i(16))
sorted_eligible_ids = eligible_entry_ids.map(&:to_i).sort
eligible_ids_hash = Digest::SHA256.hexdigest(sorted_eligible_ids.join("\n"))

total_to_select = number_of_winners + number_of_reserve_winners
selected = sorted_eligible_ids.shuffle(random: rng).first(total_to_select)

# First N items are winners 1..N in order; next M items are reserves 1..M.
```

Three properties:

Cryptographically strong entropy in the seed. The seed comes from SecureRandom.hex(32), which reads from the operating system's CSPRNG (/dev/urandom or getrandom(2) on Linux, BCryptGenRandom on Windows).

256 bits of entropy means there are 2^256 possible seeds -- practically unguessable in advance, and the seed bits themselves are statistically indistinguishable from uniform. The shuffle's uniformity is a separate property (see the next bullet); the two claims do not depend on each other.

Uniform random selection. Array#shuffle is a Fisher-Yates implementation backed by the supplied Random instance. Each permutation of n elements has probability 1/n!; each entry has probability k/n of appearing in the first k positions. Ruby's Random is Mersenne Twister (MT19937), not a CSPRNG -- this is intentional and irrelevant for fairness. CSPRNG status only matters when an adversary can observe partial RNG output and predict the rest; in our model the seed is the unguessable input and the entire shuffle output is published after the draw, so there is nothing to predict. Fisher-Yates uniformity is a property of the algorithm given an unbiased rand(n), not of the RNG's cryptographic strength.

Reproducible. Given the seed and the canonical sorted input list, the shuffle is deterministic. Anyone -- auditor, regulator, contestant -- can re-run the snippet and obtain the same winners. The Ruby version is recorded on the contest (draw_ruby_version) so future inspectors know which Array#shuffle implementation produced the result. Cross-language reproduction (Python, JavaScript, Go) is theoretically possible but requires re-implementing Ruby's specific Mersenne Twister seeding (init_by_array over the 64-bit-chunk decomposition of the integer seed) and Ruby's specific rand(n) rejection-sampling rules -- none of which come for free from another language's standard random library. The practical guidance for verifiers is therefore "install MRI Ruby 3.0"; the /verify-fairness page walks through this. Acknowledging this constraint up front: byte-compatible reproduction is Ruby-specific, not language-agnostic.

3.2 Position Assignment Is The Shuffle

There is no second randomization step. The shuffled order is the position order: the first number_of_winners IDs become winners 1, 2, 3, ... and the next 3 × number_of_winners IDs become reserves 1, 2, 3, ... Because Fisher-Yates produces a uniformly random permutation, every position assignment is itself uniformly random; collapsing the two steps that v1 used into a single shuffle simplifies what a third-party verifier has to reproduce.

3.3 What Is Recorded on Each Draw

On every successful draw, the system writes the following columns to facebook_page_contests inside the same atomic transaction that flips the contest to drawn status:

Column	Content	Purpose
draw_seed	64-hex-char string	The 256-bit RNG seed
draw_eligible_ids	JSONB array of integers	The exact sorted ID list used as input
draw_eligible_ids_hash	64-hex-char SHA-256	Lets a verifier check the ID list was not tampered with
draw_algorithm_version	string ("v1")	Future-proofing: tells the verifier which algorithm to use
draw_ruby_version	string	Records the Ruby version under which the shuffle was originally executed

3.4 Atomicity

The seeded shuffle itself is a pure in-memory computation (no DB writes), so it executes before the transaction opens; what's atomic is the persistence of its result:

```
# Pre-transaction (deterministic given the seed; no DB state involved):
seed_hex           = SecureRandom.hex(32)
sorted_eligible_ids = eligible_entry_ids.map(&:to_i).sort
eligible_ids_hash   = Digest::SHA256.hexdigest(sorted_eligible_ids.join("\n"))
selected_ids        = sorted_eligible_ids.shuffle(random: Random.new(seed_hex.to_i(16))).first(total)

ActiveRecord::Base.transaction do
# 1. Mark non-eligible entries as disqualified (with reason)
# 2. Reset winner-state columns on eligible entries (clears stale flags from any previous attempt)
# 3. Write is_winner / winner_position / reserve_winner_number on the selected IDs
# 4. Persist seed, eligible-ids list, hash, algorithm version, ruby version on the contest
# 5. Flip contest status to "drawn"
end
```

If any step inside the transaction fails (database error, constraint violation, trigger rejection), the entire transaction is rolled back. There is no partial state -- either every transcript column and every winner is committed atomically, or none are. If the process crashes between the in-memory shuffle and the BEGIN, no state is touched at all and the draw can simply be retried (the new attempt generates a fresh seed; the old one was never observed by anyone).

4. One Entry Per Person

4.1 Database Constraint

A unique index enforces that each person can have at most one entry per contest:

```
CREATE UNIQUE INDEX idx_contest_entries_unique_profile
ON contest_entries (facebook_page_contest_id, facebook_profile_id);
```

If the same person comments multiple times, the system uses PostgreSQL UPSERT to keep only the latest comment data while maintaining a single entry record:

```
ContestEntry.upsert_all(records,
unique_by: [:facebook_page_contest_id, :facebook_profile_id])
```

4.2 Implication for Fairness

A person who comments 100 times has exactly the same probability of winning as a person who comments once. The database constraint makes it structurally impossible for multiple comments to create multiple chances.

5. Eligibility Filtering

5.1 Rule Types

Contest organizers define eligibility rules before the draw. These rules are applied uniformly to all entries. The application distinguishes between automatically enforced rules (executed in code, with failures producing a disqualification_reason) and manual-verification rules (where the Facebook/Instagram Graph API does not expose reliable data; these are recorded as filter_warnings on the contest but do not disqualify anyone automatically):

Rule	Enforcement	Data Source
Email required (Facebook only)	Auto -- SQL: email IS NOT NULL AND email <> ''	Comment text from API
Date range	Auto -- SQL: (comment_time IS NULL OR comment_time >= starts_at) AND (comment_time IS NULL OR comment_time <= ends_at) -- NULL timestamps are included (see 5.2)	Comment timestamp from API
Keyword required	Auto -- SQL: LOWER(comment_text) LIKE '%keyword%' with sanitize_sql_like	Comment text from API
Minimum tags	Auto -- SQL: jsonb_array_length(COALESCE(message_tags, '[]'::jsonb)) >= N	Message tags from API
Exclude page admins (Facebook only)	Auto -- API: admin list from Facebook Graph API	Facebook API
Exclude specific users	Auto -- SQL: facebook_profile_id NOT IN (...)	User-provided list
Exclude previous winners	Auto -- SQL: cross-contest query on winner entries	Database
Must like post	Manual only -- the Facebook Graph API does not reliably expose liker IDs (only users who have authorized the app are visible); enforcement is flagged as a warning and left to the organizer to verify	N/A
Must share post	Manual only -- Facebook's sharedposts endpoint no longer returns data for page posts; enforcement is flagged as a warning	N/A

Must follow page	Manual only -- follower lists are not reliably exposed by the Graph API; enforcement is flagged as a warning	N/A
Exclude business accounts	Not currently enforced -- the column exists on the contest and is settable in the form, but no pipeline step reads it and no filter_warnings entry is emitted. Treat as a no-op for now; the organizer must use "exclude specific users" if they want to remove individual business pages. See follow-ups for the planned remediation.	N/A

The three manual-verification rules are surfaced on the contest page as a filter_warnings notice to the organizer so the organizer knows the system could not verify them automatically. This is material to any fairness claim: if a court is asked whether "all commenters who liked the post" were included in a draw, the honest answer is that the system did not filter out non-likers and, on contests where the organizer required a like, non-likers remain in the pool unless the organizer manually excludes them via the "exclude specific users" rule.

5.2 Null and Missing Data

- NULL comment timestamps are included intentionally in the date-range filter. Some Graph API responses (notably Instagram replies and certain legacy records) omit created_time; excluding those entries would unfairly disqualify legitimate commenters whose timestamp was not returned by the API.
- Keyword matching is case-insensitive and safely escaped via ActiveRecord::Base.sanitize_sql_like so that special LIKE metacharacters in user-supplied keywords are treated literally.

5.3 Rule Application

All rules are applied programmatically. The contest organizer cannot manually include or exclude specific entries after the draw begins. Rules are defined at contest creation time and locked once the contest enters processing status (editing is blocked by the controller for any status other than pending or failed). If a draw fails before any winner is selected, the contest reverts to failed and the organizer may edit rules before retrying; because no winners were ever recorded, this does not modify a drawn contest's rules.

5.4 Disqualification Transparency

Every disqualified entry carries a disqualified: true flag and a disqualification_reason. There are exactly two sources that populate this column, and they have different provenance, different authorship, and different liability implications:

System-generated reason for filter-rejected entries. When the seeded shuffle runs, every entry that did not pass the configured eligibility filters is updated in a single batched write with one generic reason -- the localized equivalent of "Did not meet contest requirements" (the canonical English string lives at services.winner_selector.did_not_meet in config/locales/en.yml). The same text is written to every filter-rejected entry on a given contest. The system intentionally does not attempt to ascribe a specific failing rule per entry, because a single entry can fail multiple rules simultaneously (e.g., missing keyword and outside date range) and singling out one would be misleading. The authoritative source of truth for why a given entry was rejected is the configured rule set on the contest itself -- every active rule and every filter_warnings entry (§5.1) is publicly visible on the results page, so a participant or auditor can compare an entry's comment against the rule set to deduce which rule(s) it failed.

Organizer-supplied reason for redrawn winners. When the contest organizer triggers a redraw under §7.3, the free-text reason they enter through the redraw form is written to the displaced winner's disqualification_reason. That text is authored entirely by the organizer, at the organizer's own discretion, at the moment of the redraw. Pick a Winner prompts the organizer to record a reason and stores whatever they type; the application does not draft, suggest, validate, edit, moderate, or otherwise control the content of organizer-supplied reasons. Any dispute, claim, or complaint over the wording, accuracy, fairness, or tone of an organizer-authored reason -- including any claim that it misrepresents why a winner was removed -- is a matter between the contest organizer and the affected participant. Pick a Winner is not liable for the content of organizer-authored disqualification text and accepts no responsibility for resolving such disputes. The EULA states the corresponding contractual position in §12.5.1.

This data is visible to the contest owner, on the public results page, and in the CSV export. Once the contest is drawn, both the disqualification flag and the reason column are DB-immutable (§7.2); the single audited carve-out is the redraw_winner! path, which is the only mechanism by which a disqualification_reason row can change

after the draw -- and it changes precisely because the organizer just typed a new value.

6. Concurrency and Manipulation Prevention

6.1 Advisory Locks

Each draw acquires a PostgreSQL advisory lock scoped to the contest ID:

```
SELECT pg_try_advisory_lock(999001, contest_id);
```

This prevents two concurrent draws of the same contest. If a lock is already held, the second attempt is silently skipped. The lock is released in an ensure block (guaranteed cleanup). A separate transaction-scoped advisory lock (AdvisoryLocks::AUDIT_EVENT_CHAIN, namespace 999_005) serializes audit-event chain insertion per contest so concurrent writers cannot create two events sharing the same predecessor (see §8.2).

6.2 State Machine

The contest follows a strict state machine with validated transitions (invalid transitions raise ContestStateMachine::InvalidTransitionError):

```
pending          -> processing
processing       -> drawn | pending (cancel) | payment_required | failed
payment_required -> processing
drawn            -> completed | failed
failed           -> processing
completed        -> (terminal)
```

The transition from processing to drawn can only occur inside the SelectWinners pipeline step, within the atomic transaction that also writes is_winner, winner_position, the seed, and the eligible-ID transcript. There is no API endpoint or user action that can directly set a contest to drawn status.

6.3 Idempotency

The draw job checks winners_drawn? before processing. If winners have already been selected (by a previous run), the job exits without action. This prevents accidental re-draws.

6.4 Cancellation

A contest organizer can cancel a draw while it's processing. Cancellation sets cancelled_at on the contest record, and the pipeline checks for this between batches during comment fetching. Cancellation returns the contest to pending status -- it does not select or modify winners.

7. Winner Immutability After Draw

7.1 The Threat Model

Once winners are recorded, the value of any "fair draw" claim depends on the recorded winners remaining unchanged. The v1 design (before 2026-05-27) relied on the absence of an in-app endpoint for modifying winner columns; an operator with Rails console access could nonetheless flip is_winner, winner_position, or disqualified via update!, update_columns, or raw SQL with no record left behind. The hardening landed 2026-05-27 closes this gap at the database level.

7.2 PostgreSQL Trigger

A BEFORE UPDATE trigger on contest_entries runs on every UPDATE statement. If the joined contest is in drawn or completed status, the trigger raises if any of the following columns would change:

- is_winner
- winner_position
- reserve_winner_number
- disqualified

- disqualification_reason

Each column defends a distinct attack surface, which is why all five are locked together rather than just the headline is_winner flag:

- is_winner -- was X picked at all? Flipping this is the obvious rigging attack.
- winner_position -- which prize tier did X win? Swapping positions between two existing winners (so the 1st-prize winner becomes the 2nd-prize winner and vice versa) leaves both is_winner flags true and would not be caught by guarding is_winner alone.
- reserve_winner_number -- which reserve is the next call-up if a primary winner doesn't respond? Re-ordering reserves silently after the draw is its own attack class.
- disqualified and disqualification_reason -- was X excluded, and on what stated grounds? Retroactively disqualifying a winner so a redraw can promote a chosen reserve, or rewriting a stated reason to fit a chosen narrative, would both be inert against guards on the winner columns alone.

The trigger fires before the row is rewritten, so the offending UPDATE is rejected by PostgreSQL itself. update!, update_columns, raw SQL, and update_all all hit the same trigger. The operator can disable the trigger only by running DROP TRIGGER ... or SET session_replication_role = replica, both of which require PostgreSQL superuser privileges and are logged by PostgreSQL itself.

7.3 The One Legitimate Exception: redraw_winner!

A redraw -- replacing a non-responsive primary winner with the next reserve -- is the only legitimate post-draw modification. The redraw_winner! model method holds a row lock on the contest, sets a transaction-scoped GUC override (SET LOCAL pick_a_winner.allow_redraw = 'on') that the trigger reads as an allow-list, then writes the change. The override:

- Is scoped to a single transaction by SET LOCAL
- Is explicitly RESET in an ensure block, so it cannot leak even when redraw_winner! is called from within an outer transaction
- Records a redraw_winner audit event (position, reason, old/new winner profile IDs; §8) whenever the caller passes performed_by:. The only production caller -- the redraw_winner controller action -- always passes current_user, so every production redraw is audited. A caller that omits performed_by: (e.g., a Rails-console operation or a future automated path) still uses the override and so still completes the winner swap, but skips the audit-event write; see follow-ups for the planned tightening that makes performed_by: mandatory and closes this gap.

The free-text reason written to the audit event metadata -- and copied into the displaced winner's disqualification_reason -- is authored by the contest organizer through the redraw form; Pick a Winner does not generate, edit, or moderate this text. See §5.4 for the corresponding fairness-disclosure framing and EULA §12.5.1 / §12.7 for the contractual position.

There is no other code path that sets the override.

7.4 What Replaces Whom

Replacement selection is deterministic first, random second, in a way that preserves fairness:

1. The original draw (§3) randomly selects 3 × number_of_winners reserve entries in the same atomic transaction as the primary winners. Each reserve is assigned a reserve_winner_number in an order that was itself uniformly randomized at draw time. Reserves are persisted before any redraw is possible, so the replacement sequence is fixed in the audit trail from the moment the draw completes.
2. On a redraw, the system first picks the lowest-numbered remaining reserve (ORDER BY reserve_winner_number ASC LIMIT 1 FOR UPDATE). This is deterministic given the draw result, but the ordering itself originated from a uniform random process recorded in the published transcript.
3. Only if no reserves remain does the system fall back to ORDER BY RANDOM() over the pool of non-winners who passed the original filters. (This non-reproducible fallback is invoked only after all 3N reserves are exhausted, which is rare in practice.)

In both paths, the organizer cannot choose a specific replacement; they can only trigger a redraw for a given position with a mandatory reason. The row-level FOR UPDATE lock prevents two concurrent redraws from selecting the same replacement.

8. Append-Only, Hash-Chained Audit Log

8.1 What Is Recorded

Every significant action creates a ContestAuditEvent record:

Action	Metadata Stored
draw_winners	winner_count, total_entries, drawn_at, duration_ms, draw_seed, draw_algorithm_version, draw_eligible_ids_hash
redraw_winner	position, reason, old_winner_profile_id, old_winner_name, new_winner_profile_id, new_winner_name
cancel	cancelled_at
winner_notified	winner_id, winner_position, facebook_comment_id, facebook_reply_id, attempt_number, notified_at
winner_notification_failed	winner_id, winner_position, facebook_comment_id, attempt_number, error_class, error_message, attempted_at
winner_marked_dmed	winner_id, winner_position, marked_at
winner_unmarked_dmed	winner_id, winner_position, previous_notified_at, unmarked_at
winner_email_delivered	winner_id, winner_position, delivered_at

Every event stores the acting user (via belongs_to :user), a created_at timestamp written by the database, and the two hash-chain columns introduced in §8.2. For draws triggered by the scheduled AutoDrawContestsJob (rather than a manual "Draw Winners" click), the recorded user is the contest owner -- the system has no separate actor identity. A reader inspecting the audit log should therefore interpret the user_id field as identifying the contest's responsible party, not necessarily the human who initiated this specific event.

8.2 Per-Event Cryptographic Chaining

Each ContestAuditEvent row carries two columns added 2026-05-27:

- previous_hash -- the entry_hash of the immediately prior event in this contest's chain, or a fixed genesis hash ("0" * 64) for the first event
- entry_hash -- SHA-256 over a canonical JSON serialization of {previous_hash, action, facebook_page_contest_id, user_id, metadata, created_at}, with deep-sorted JSON keys and microsecond-precision UTC timestamps for byte-stable input

A before_create callback computes both values inside a pg_advisory_xact_lock(AUDIT_EVENT_CHAIN, contest_id) to serialize concurrent inserts per contest. Each contest is an independent ledger. Result: any silent modification or deletion of an audit row leaves either the row's own entry_hash mismatched against its computed value, or the next row's previous_hash mismatched against the modified row's entry_hash -- and both cases are detected by the verifier walking the chain.

A global UNIQUE index on contest_audit_events.entry_hash provides an additional collision/replay barrier across all contests: even if an attacker disabled the trigger, they could not insert a row reusing any prior entry_hash value anywhere in the table. Combined with microsecond-precision created_at and the per-contest advisory lock, this makes accidental or adversarial hash reuse a database-level constraint violation rather than a silently-accepted write.

8.3 PostgreSQL Triggers

Two triggers enforce append-only behavior at the database level:

- BEFORE UPDATE on contest_audit_events -- always raises. There is no GUC override; updates are never permitted.

- BEFORE DELETE on contest_audit_events -- raises by default; permits the delete only under SET LOCAL pick_a_winner.allow_audit_purge = 'on', set by FacebookPageContest#before_destroy (prepend: true) and User#before_destroy (prepend: true) so that legitimate parent-cascade destruction can complete.

The deliberate carve-out for parent cascade is documented in lib/database_triggers.rb and is the single exception to "audit events are forever".

The practical implication is that a contest's audit chain does not survive the deletion of the contest record itself: a User#destroy cascade, an admin cleanup of an abandoned account, a demo-user reseed, or an explicit contest deletion will take that contest's audit events with it (the cascade flips the GUC override and the trigger then permits the DELETE). Refusing the cascade would have left orphan audit events pointing at a vanished facebook_page_contest_id, which is its own integrity problem. The mitigation against an adversarial use of this carve-out is external rather than internal: any party who pulled an audit-log.json snapshot before the destruction retains an independent copy of the chain, and any Bitcoin-confirmed .ots proof downloaded before then continues to verify against Bitcoin regardless of what the operator's database now contains. See §12.6 for the bounded threat description and the planned external-snapshot-store follow-up that would close the residual gap.

8.4 Verification

Two surfaces:

- ContestAuditEvent.verify_chain!(contest) -- walks the chain in id order for one contest, raising ContestAuditEvent::ChainBroken on the first inconsistency
- rake audit:verify -- walks every contest's chain and reports breaks; exits non-zero on failure

AdminDailyDigestJob runs verify_chain! for every contest active in the last 7 days. On any break it records a critical SystemAlert with 23-hour dedup, surfacing the failure to admins.

8.5 Per-Winner Delivery Receipts

Added 2026-05-28. Every winner notification leaves a per-winner receipt and a corresponding audit event so a verifier can confirm the operator both selected and notified each winner.

Three notification modes, three proof strengths:

- Facebook comment-reply mode (no_email_facebook?) -- strongest. The system posts a public reply to each winner's original comment via Graph put_comment. The returned reply ID is persisted on the contest_entries row (facebook_reply_id) and a winner_notified audit event is written with the reply ID in its metadata. The public results page renders a "Verified on Facebook" link per winner that deep-links to the actual reply on the post -- anyone can click and see the proof live on Facebook itself, outside the application's database. If a per-winner call fails, a winner_notification_failed event records the error class and message; the contest stays in drawn status until every winner has a successful receipt.
- Facebook email mode (require_email?) -- per-winner notified_at is stamped at email-enqueue time. An ActionMailer observer (WinnerNotificationObserver) writes a winner_email_delivered audit event when the SMTP layer actually accepts the message -- that event is the proof of delivery and is hash-chained like every other event. If SMTP fails, the observer does not fire; the gap between enqueue (stamped) and delivery (not stamped) is visible in the audit log. The recipient address is intentionally not included in the event metadata to keep email PII out of the public audit chain.
- Instagram mode (instagram?) -- owner-attested. Facebook's Graph API does not expose Instagram DM send. The owner manually DMs each winner outside the app, then clicks a per-winner "Mark DMed" button. The click writes a winner_marked_dmed audit event identifying the winner and the marking time. This is self-attestation, not an externally verifiable receipt -- but it is hash-chained and timestamped, so any later edit by the operator would break the chain in the same way a tampered draw event would.

Idempotency. The Facebook comment-reply replier is keyed on facebook_reply_id IS NULL -- re-running a partially-failed batch only attempts winners who don't yet have a receipt. A successful reply is never posted twice from the application's side. The narrow remaining duplicate-reply window (Facebook accepted but the response was lost in transit) is documented in the replier's docstring and in tmp/followups.md as an accepted v1

limitation.

Public verification surface. The audit-log JSON endpoint at results/:token/audit-log.json includes the new notification event types (winner_notified, winner_notification_failed, winner_marked_dmed, winner_unmarked_dmed, winner_email_delivered) in its events array with no application-level filter -- every notification appears alongside draw_winners/redraw_winner/cancel and contributes to the same hash chain. An external watchdog polling the endpoint sees notification events and notification failures in real time.

8.6 What This Catches, What It Does Not

Catches:

- UPDATE of any audit row through the application or via raw SQL (blocked at the trigger)
- DELETE of any audit row outside the GUC-overridden parent-cascade path (blocked at the trigger)
- Silent injection of a forged audit row (the chain breaks at insertion because the verifier recomputes hashes from canonical inputs)
- Reordering of events for one contest (the predecessor pointer mismatches)

Does not catch:

- A PostgreSQL superuser who disables the triggers, inserts forged rows, and recomputes every downstream hash to maintain chain consistency. This attack requires database-level privileges beyond the application's authority and would still appear in PostgreSQL's own query log. The next hardening layer -- periodically anchoring chain heads to an external public store such as a git repository or an OpenTimestamps proof -- would prevent even this attack by making forgery detectable to any outside observer.
- Tampering with the source comments fetched from Facebook/Instagram before they enter the database. The Graph API response is the source of truth and is not cryptographically signed by the platform.

9. Public Verifiability

For a reader-oriented walkthrough of every verification level (from a one-glance badge through scripted retroactive-tamper monitoring), see the dedicated Verify a Draw Yourself guide. This section describes the surfaces that guide refers to.

9.1 The Two Public Results URLs

Every drawn contest is publicly reachable at two equivalent URLs:

- /facebook_page_contests/<slug> -- served by FacebookPageContestsController#show. This is the SEO-friendly URL the operator typically posts on social media; it's what shows up in Facebook/Instagram link previews and search results. No token in the URL.
- /results/<token>/<slug> -- served by PublicResultsController#show. The token-keyed canonical URL. Identical content; included primarily for direct linking from the audit-log JSON and for verifiers who want a URL that is unambiguous about which contest a token refers to.

Both pages render the same compact verification strip for contests drawn on or after 2026-05-27. The strip shows:

- A status badge: Verified, Redrawn since draw, Tampering detected, or Pre-hardening draw
- A one-line claim ("Independently verifiable by any third party")
- A "How to verify yourself ->" link that lands on the per-contest verify helper described in §9.2

Hex transcripts (seed, hash, algorithm version, Ruby version), download links, and the Ruby snippet are not rendered inline on either URL. This is a deliberate UX choice: the operator's shareable URL stays clean and readable, while a single click takes a verifier to a dedicated page where every technical artifact is collected. The badge itself is computed server-side by DrawVerifier on every render; there is no cached representation that could mask tampering.

For contests drawn before 2026-05-27, the strip shows a "Pre-hardening draw -- not independently verifiable" notice instead of the Verified badge. This is the honest disclosure; see §12.

9.2 The Per-Contest Verify Helper

A third public URL gathers everything a verifier needs for one specific contest:

/results/<token>/verify (served by PublicResultsController#verify)

The page renders:

- The badge again, with explicit "Verified by re-running the seeded shuffle" prose
- Algorithm version (v1) and the Ruby version under which the draw ran
- The full 256-bit random seed (hex)
- The SHA-256 of the sorted eligible-IDs list
- A download link for eligible-ids.txt
- A link to the audit-log JSON (§9.6)
- The latest Bitcoin-anchor proof URL (if anchored) with the Bitcoin block height
- Cross-links to the step-by-step guide (/verify-fairness) and this document

The "How to verify yourself" link on the compact strip (on either of the two public URLs above) lands here. Cache lifetime is 10 minutes so pending -> Bitcoin-confirmed transitions become visible quickly.

9.3 How to Re-Run a Draw

A third party can verify any post-hardening contest in three steps:

1. Visit /results/<token>/verify (or click "How to verify yourself ->" on any results page).
2. From there, download eligible-ids.txt and copy the published seed.
3. Run in IRB (or any Ruby >= 3.x):

```
require "digest"
seed = "PASTE_THE_PUBLISHED_SEED_HERE"
ids  = File.read("eligible-ids.txt").split("\n").map(&:to_i).sort

# (1) Sanity check the input: must equal the published SHA-256.
Digest::SHA256.hexdigest(ids.join("\n"))

# (2) Re-run the shuffle. First N items are winners 1..N (in order);
#     the next 3N items are reserves 1..3N.
ids.shuffle(random: Random.new(seed.to_i(16))).first(N_winners + N_reserves)
```

This snippet is character-identical to the one on /verify-fairness Level 2 -- copying from either lands at the same code.

If the hash matches AND the shuffled winners match what the page shows, the draw is verified. If either fails, the draw should be considered tampered. The dedicated /verify-fairness guide walks through this in detail as Level 2.

9.4 What Else Is Public

On the results pages (§9.1) and the per-contest verify helper (§9.2), the publicly displayed content includes:

- Contest title, description, dates, prize descriptions
- Winners, each shown with position, profile name, comment text, and obfuscated email
- Disqualified entries with their reason
- Up to 1,000 participants in chronological order (lazy-paginated; the full untruncated list is always available to the contest owner via the CSV export described in §9.5)

All email addresses on the public page are obfuscated; unobfuscated emails appear only in the owner-only CSV export.

9.5 CSV Export

The contest owner (and only the owner -- non-owners receive 404, and the export is enabled only once the contest has reached completed status) can export the full entry list as a CSV file containing:

- Entry number, participant name, profile ID, email
- Comment text
- Winner status (Yes/No) and winner_position
- Disqualification status (Yes/No) and disqualification reason

The CSV is streamed row-by-row from the database in chronological comment order and contains every entry with no row cap, providing a complete portable record of the draw. A 5-minute per-user cooldown applies to non-admin exports to mitigate scraping.

Because the CSV is owner-only, an independent verifier (anyone other than the contest owner) who wants the complete pool -- beyond the 1,000-entry public pagination on the results page -- reconstructs it from the public eligible-ids.txt plus the original contest post on Facebook/Instagram. The CSV is therefore the operator's full-record artifact; the eligible-ids file is the public's. This split is intentional: the CSV carries unobfuscated emails which the operator legitimately needs for prize delivery; the public path carries only the information needed to reproduce and audit the draw.

9.6 Audit Chain API

A read-only JSON endpoint exposes the contest's full hash-chained audit history (every event in chain order, with previous_hash, entry_hash, the canonical metadata, and the actor's internal user id). The endpoint is mounted at two equivalent paths so a reader who landed on either of the two public results URLs (§9.1) can reach the audit log by appending one path segment to the URL already in their address bar:

- GET /results/:token/audit-log.json -- paired with the canonical token-keyed results page.
- GET /facebook_page_contests/:slug/audit-log.json -- paired with the SEO-friendly slug results page; this is the URL operators typically share.

Both URLs are served by the same ContestAuditChainSerializer and return byte-identical JSON (the contest_token, contest_slug, events, anchors, and verification fields are populated the same way regardless of which path the request arrived on). A controller test pins this parity (test/controllers/facebook_page_contests_controller_test.rb -- "slug audit-log.json payload matches the token audit-log.json payload"). Every request runs ContestAuditEvent.verify_chain! server-side and returns a chain_status field that is either the literal string "ok" or the ChainBroken exception message naming the offending event -- e.g., "previous_hash mismatch at event #5 (expected ..., got ...)" or "entry_hash mismatch at event #5". The response is sent with Cache-Control: no-store so a tamper alert is visible the instant a poller pulls.

The two-URL surface is purely a discoverability convenience. The data, gating (winners_drawn? only), caching policy (no-store), and authentication posture (none) are identical on both. An anchor's proof_url field in the JSON still points at the token-keyed audit-anchors/:id.ots endpoint regardless of which path served the JSON, so a verifier walking links from the response always reaches the same proof bytes.

What this endpoint accomplishes:

- External tamper detection. Anyone (contestants, journalists, regulators, the operator themselves) can poll the endpoint, snapshot the returned chain, and diff successive snapshots. Any deletion, insertion, or modification of an event between snapshots is visible to the holder of the prior snapshot -- regardless of whether the on-disk DB still passes its own internal chain check.
- Real-time chain status. Because each request recomputes chain_status, a tamper is detected by an external watchdog the moment it polls, rather than waiting for the next AdminDailyDigestJob run.
- Cryptographic recomputation enabled. The response includes a verification block (algorithm, payload field order, timestamp format, genesis hash, JSON canonicalization rule). A motivated verifier can rebuild the canonical payload from the exposed fields and recompute every entry_hash independently.
- Foundation for external anchoring. External chain anchoring (the next followup) commits an entry_hash to an out-of-band public store. The audit-log endpoint provides the chain history those anchors prove the

position of.

What this endpoint does NOT achieve:

- Does not prevent forgery. A PostgreSQL superuser who disables the triggers and rewrites both contest_entries and contest_audit_events (recomputing every downstream hash) will produce a clean chain that the endpoint reports as "ok". The endpoint only makes forgery detectable -- and only to parties who held a snapshot from before the rewrite.
- PII surface unchanged. Audit events store action, position, profile IDs and names (already public on the results page), and the operator's internal user_id (an opaque integer required for hash recomputation). No new personal data is exposed.

9.7 Bitcoin-anchored timestamps (OpenTimestamps)

Every contest's audit-chain head is periodically committed to the Bitcoin blockchain via OpenTimestamps. An hourly background job submits a digest of (namespace || contest_id || latest_entry_hash) to several OpenTimestamps calendar servers; a six-hourly job upgrades each pending proof to a full Bitcoin-anchored proof once a Bitcoin block has confirmed (typically within ~6 hours of submission).

The result is exposed in two places:

- The audit-log.json response (§9.6) gains an anchors array. Each entry carries the anchored entry hash, submission and upgrade timestamps, the Bitcoin block height (once upgraded), and a download URL for the raw proof bytes.
- GET /results/:token/audit-anchors/:id.ots serves the proof bytes directly. Anyone with the standard OpenTimestamps CLI can run ots verify -d <digest> <proof.ots> against the proof to confirm Bitcoin anchoring without trusting Pick a Winner, OpenTimestamps' calendar servers, or anyone but the Bitcoin blockchain itself.

Why this matters: §12 names the PostgreSQL-superuser threat as the residual limit of the application-layer hardening. With Bitcoin anchoring in place, any chain event whose anchor has been upgraded (~6 hours after the event) is structurally protected against rewriting -- the operator can rewrite their own database, but cannot rewrite Bitcoin's history. The detailed verification workflow is on the Verify a Draw Yourself guide as Level 5.

9.8 Data Retention

- Contest data is retained for the lifetime of the account
- Audit events are retained for the lifetime of the contest (subject to the parent-cascade exception in §8.3)
- Acknowledged system alerts are retained for 90 days
- Deleting a contest permanently removes its associated rows (cascade)

10. What the Contest Organizer Can and Cannot Do

CAN do (before the draw):

- Configure eligibility rules: auto-enforced rules (tags, keywords, date range, email-required, exclude-specific-users, exclude-previous-winners, exclude-page-admins) and manual-verification rules (must-like / must-share / must-follow -- informed these cannot be automatically enforced; see §5.1)
- Set the number of winners (1-12); the system automatically draws 3 × number_of_winners reserves alongside the primary winners
- Exclude specific Facebook/Instagram profile IDs
- Exclude previous winners from other contests on the same page
- Schedule an automatic draw at a specific time (scheduled_draw_at) or rely on end-of-contest auto-draw

CANNOT do (at any point):

- Select or influence which specific person wins
- Modify the eligible pool after the draw begins (edit is blocked once status leaves pending / failed)
- Skip the randomization step
- Reuse a seed across contests (each draw generates its own from the OS CSPRNG)
- Run the draw while another draw is in progress (advisory lock)
- Choose a specific replacement during a redraw (reserve order is fixed at draw time, fallback is random)
- Modify or delete audit trail entries through the application (DB triggers; §8.3)
- Modify winner/disqualification columns on a drawn contest (DB trigger; §7.2)
- Create duplicate entries for a favored participant (database unique constraint)

CAN do (after the draw):

- Redraw a specific position with a mandatory reason (fully audited; replacement drawn from the pre-committed reserves first; §7.3)
- Export the full entry list as CSV (once the contest is completed; owner-only; 5-minute cooldown)
- Share the public results page (which now includes the verification surface; §9.1)

11. Technical Guarantees Summary

Property	Mechanism	Bypassable?
Equal probability per entry	Array#shuffle (Fisher-Yates) over a seeded Random	No -- algorithmic, no user input
Reproducible draw	Seed + sorted eligible-IDs list + their SHA-256 stored on the contest; published on the public page	No -- anyone can re-run the shuffle and confirm
One entry per person	Database unique index on (facebook_page_contest_id, facebook_profile_id)	No -- enforced at DB level via upsert_all
No concurrent draws	PostgreSQL advisory lock (namespace 999_001, contest_id)	No -- cooperative lock acquired before the pipeline runs
Atomic winner + reserve selection + transcript	Single ActiveRecord::Base.transaction	No -- all or nothing
Winner state immutable post-draw	BEFORE UPDATE PostgreSQL trigger on contest_entries, gated on contest status	Only via DB superuser disabling the trigger (logged by PostgreSQL); the in-app redraw_winner! GUC override is the single audited exception
Audit trail append-only	BEFORE UPDATE / DELETE PostgreSQL trigger on contest_audit_events	Updates: never. Deletes: only via the documented parent-cascade GUC override
Audit trail tamper-evident	SHA-256 hash chain per contest; verified daily by AdminDailyDigestJob	A DB superuser who rewrites every downstream hash can forge -- but for any chain event past its Bitcoin anchor (~6h confirmation window), forgery would also require rewriting Bitcoin
Audit chain Bitcoin-anchored	Hourly OpenTimestamps submission + six-hourly proof-upgrade poll (Bitcoin itself confirms calendar batches every ~10 min, independently of our poll cadence); .ots proof downloadable per anchor	No -- anyone with the proof can run ots verify against the Bitcoin blockchain itself
Public verifiability	Two equivalent public URLs (/facebook_page_contests/<slug> and /results/<token>/<slug>) show the verification badge; the per-contest verify helper at /results/<token>/verify exposes seed, hash, downloadable IDs, audit log link, and latest anchor proof. The audit-log JSON is itself reachable at two equivalent paths (token-keyed and slug-keyed, §9.6)	Anyone with the link can independently confirm winners; no auth required
No manual winner selection	Pipeline automation with no UI affordance	No UI, no controller endpoint, and no DB write path that does not pass through the seeded shuffle
Disqualification transparency	Reason stored per entry, surfaced publicly, in CSV, and immutable post-draw	Visible on results page; cannot be altered after drawn

Redraw fairness	Reserves pre-drawn at original draw time (uniformly randomized via the same seed); random non-winner fallback only when reserves are exhausted; every redraw fully audited	Organizer cannot choose replacement; row-level FOR UPDATE prevents concurrent-redraw collisions
Rule immutability after draw start	Controller blocks edit/update for any status outside pending / failed	No; a failed draw can be edited before retry, but no winners are ever recorded on a failed draw

12. Honest Limitations

The document loses credibility if it claims more than it can prove. Each limitation below is named explicitly so a regulator, court, or skeptical contestant can decide whether the residual risk is acceptable for their purpose.

12.1 Pre-hardening contests (real, bounded by date)

Contests drawn before 2026-05-27 do not have a recorded seed and cannot be retroactively re-run. The public results page surfaces this with a "Pre-hardening draw -- not independently verifiable" notice. Their winners are real -- they were drawn from the same uniform-random pipeline -- but a third party cannot recompute them. Bound: strict cutoff date; affects only legacy contests; every contest drawn after 2026-05-27 is verifiable.

12.2 Pending-anchor window (real, ~6 hours per event)

An audit event written less than ~6 hours ago is anchored to the OpenTimestamps calendar servers but has not yet been confirmed in a Bitcoin block. During this window, a determined PostgreSQL superuser who disabled the triggers could in principle rewrite the event and would only need to fool the OTS calendar servers (not Bitcoin) to forge a clean ledger. Once Bitcoin confirms the anchor (typically 1-6 hours after submission), the event is structurally protected -- rewriting it would require rewriting Bitcoin's blockchain, which is not under any operator's control. Bound: affects only events created in the last few hours; clears automatically as Bitcoin confirms. The badge on the per-contest page distinguishes Bitcoin-confirmed anchors (block height shown) from pending ones.

12.3 Source data integrity (real, structurally unavoidable)

The eligible commenter pool is fetched from Facebook/Instagram Graph APIs, and the API response is not cryptographically signed by the platform. The system can prove what it drew from -- every byte that fed into the seeded shuffle is recorded in draw_eligible_ids -- but cannot independently prove that input matched what was publicly posted. The Facebook/Instagram post itself remains the source of truth for the comment list. Bound: this is a property of the upstream platforms, not of Pick a Winner; the only fix would require Meta to start signing API responses.

12.4 Redraws (real, fully audited)

A redraw_winner! call replaces a primary winner with a reserve. The original draw transcript still verifies as correct (the recorded seed + eligible IDs reproduce the original winners); the public page shows a "Redrawn since draw" badge and the redraw is fully audited with position, reason, and old/new winner profile IDs. A verifier must consult the audit log to understand the current winner set after a redraw. Bound: every redraw is visible, audited, and chained into the same hash chain as the original draw.

12.5 Reserves-exhausted redraw fallback (real, rare in practice)

§7.4's last-resort ORDER BY RANDOM() fallback for redraws (only triggered after all 3N reserves are exhausted) is not reproducible from the recorded transcript. This fallback is rare in practice -- it requires the operator to redraw past every pre-committed reserve -- and is fully audited. Bound: flagged as a separate audit-event metadata field so a verifier can identify which winners (if any) came from the fallback path.

12.6 Contest destruction takes the audit chain with it (real, bounded by external snapshots)

Deleting a FacebookPageContest cascades to its contest_audit_events rows via the pick_a_winner.allow_audit_purge GUC override (see §8.3); the same cascade also removes the chain_anchors rows that store the local copies of the OpenTimestamps proofs. After destruction, no in-app record of the audit chain or its anchors remains. What survives is what third parties downloaded earlier: audit-log.json snapshots

(the full chain history), and .ots proof bytes (which continue to verify against Bitcoin's history regardless of what our database now contains). An external watchdog polling audit-log.json on any schedule retains a complete, structurally-tamper-evident copy; a Bitcoin-confirmed proof saved by such a watchdog continues to attest the existence of specific chain heads at specific block heights even after the underlying contest record is gone. Bound: the residual evidence lives outside the operator's reach by design. Mitigation roadmap: the "External transcript snapshot store" project follow-up writes a write-only copy of every transcript to an out-of-band store before destruction is possible; landing it closes the in-app gap so external snapshots become a redundancy rather than the only line of defense.

12.7 What is no longer a limitation

Items previously named in this section that are now mitigated:

- "A PostgreSQL superuser can produce an internally consistent forged ledger" -- closed by Bitcoin anchoring for events older than ~6 hours (§9.7). The application-layer hardening alone never defeated this threat; the Bitcoin commitment does.
- "External chain-head anchoring is on the roadmap" -- landed 2026-05-27. See §9.7 and §15 (threat model).

13. Source Code References

All code is available for independent audit:

Component	File Path
Winner + reserve selection (seeded shuffle, transcript persistence)	app/services/winner_selector.rb
Draw verifier (re-runs the shuffle and compares)	app/services/draw_verifier.rb
Entry filtering (auto + manual-verification warnings)	app/services/contest_entry_filter.rb
Comment fetching (upsert unique by profile)	app/services/contest_pipeline/fetch_comments.rb
Pipeline orchestration	app/services/facebook_contest_service.rb
Pipeline step -- select winners	app/services/contest_pipeline/select_winners.rb
Draw job (advisory locks, circuit breaker, audit event)	app/jobs/draw_winners_job.rb
Daily chain verification	app/jobs/admin_daily_digest_job.rb
Advisory lock namespaces	config/initializers/advisory_locks.rb
Database triggers (immutability + append-only)	lib/database_triggers.rb
State machine	app/models/concerns/contest_state_machine.rb
Entry model	app/models/contest_entry.rb
Audit event model (hash chain, verify_chain!)	app/models/contest_audit_event.rb
Redraw logic (reserve-first, GUC override, audit)	app/models/facebook_page_contest.rb (redraw_winner!)
Cancellation (audit event + pending transition)	app/services/contests/cancel_draw.rb
Public results + verification surface + audit chain API (token path) + anchor proof endpoint	app/controllers/public_results_controller.rb
Audit chain API (slug path; identical payload)	app/controllers/facebook_page_contests_controller.rb (audit_log action)
Shared audit-chain JSON serializer (single source of truth for both API paths)	app/services/contest_audit_chain_serializer.rb
Verification UI partial (compact strip)	app/views/public_results/show/_verification.html.erb
Step-by-step verification guide page	app/views/pages/verify_fairness.html.erb
Per-contest verification helper page	app/views/public_results/verify.html.erb
Chain anchor model	app/models/chain_anchor.rb

OpenTimestamps CLI wrapper	app/services/ots_client.rb
Hourly anchor-submission job	app/jobs/anchor_audit_chain_job.rb
Six-hourly anchor-upgrade job (Bitcoin confirmation)	app/jobs/upgrade_chain_anchors_job.rb
Rake task -- verify a single draw	lib/tasks/draw_verify.rake
Rake task -- walk every audit chain	lib/tasks/audit_chain.rake
Recurring schedule (Solid Queue)	config/recurring.yml
CSV export service	app/services/contest_csv_export_service.rb
Minimum commenters check (paid contests)	app/services/contest_pipeline/enforce_minimum_commenters.rb

This analysis describes the system as of 2026-05-27. All claims are verifiable against the source code; the file paths in §13 are the authoritative reference. Where this document and the source disagree, the source is correct.

14. Public Verification Surfaces

Every URL below is public, requires no authentication, and is auto-populated by the system. The contest creator does no extra work to produce or maintain any of them; the only operator action that has ever existed is drawing the contest. Everything else is project automation.

14.1 Project-wide pages (apply to any contest)

URL	Returns	Audience	See
/fairness-analysis	This document, rendered as HTML	Anyone (curious, regulator, legal)	§1-§15
/fairness-analysis.pdf	This document, rendered as PDF	Auditors needing a portable artifact	§1-§15
/verify-fairness	Step-by-step verification guide across six levels of technical skill, with a glossary	Contestants, journalists, watchdogs, regulators	§9 cross-reference

14.2 Per-contest pages (one set per contest, populated by the contest's token)

For a contest whose public URL is https://pickawinner.pro/results/<token>/<slug>:

URL pattern	Returns	Content type	Audience	See
/facebook_page_contests/<slug>	SEO-friendly contest page: same content as the canonical results URL below, including the verification strip. This is the URL operators typically share on social media	text/html	Anyone (typically contestants arriving from social media)	§9.1
/results/<token>/<slug>	Canonical token-keyed results page: identical content to the SEO URL above. Linked from the audit-log JSON and from anywhere a token is the primary identifier	text/html	Anyone with the link (typically contestants, auditors following deep-links)	§9.1
/results/<token>/verify	Per-contest verification helper: gathers the badge and every URL below into one page; cross-links to /verify-fairness and /fairness-analysis	text/html	Auditors, skeptical contestants, journalists	§9.2
/results/<token>/eligible-ids.txt	The canonical sorted list of internal IDs of every comment that passed every filter, one ID per line (UTF-8, plain text)	text/plain	Anyone running Level 2 verification	§9.3
/results/<token>/audit-log.json	Full hash-chained audit log in chain order; chain_status field; anchors array; self-describing verification block. Byte-identical to the slug-keyed twin below	application/json	Anyone running Level 3+ verification; external watchdogs polling for retroactive rewrites	§9.6

/facebook_page_contests/<slug>/audit-log.json	Same audit-log JSON as the token-keyed URL above; mounted on the slug path so a verifier who arrived via the SEO results URL can reach the audit log by appending one segment	application/json	Same as above; useful when only the slug is in hand (e.g., a verifier following a social-media share)	§9.6
/results/<token>/audit-anchors/<id>.ots	Raw OpenTimestamps proof bytes for one anchor; verifier feeds these to ots verify	application/vnd.opentimestamps.v1	Anyone running Level 5 (Bitcoin) verification	§9.7

14.3 Caching policy

Endpoint	Cache-Control	Why
/fairness-analysis	public, max-age=21600 (6 hours)	Document content rarely changes
/fairness-analysis.pdf	(no cache header -- regenerated per request)	PDF re-renders are fast; freshness preferred
/verify-fairness	public, max-age=21600 (6 hours)	Static content
/facebook_page_contests/<slug>	(default Rails response, no explicit cache header)	Updated when the contest changes; mostly serves dynamic per-viewer content
/results/<token>/<slug>	HTTP-stale-aware via stale?(@contest, public: true)	Updated when the contest changes
/results/<token>/verify	public, max-age=600 (10 minutes)	Lower than the results page because anchor status changes faster
/results/<token>/eligible-ids.txt	(no cache header)	Tiny payload, deterministic, can be re-fetched freely
/results/<token>/audit-log.json	no-store	A tamper alert must be visible the moment a poller pulls
/facebook_page_contests/<slug>/audit-log.json	no-store	Same reason as the token-keyed twin -- identical serializer, identical headers
/results/<token>/audit-anchors/<id>.ots	public, max-age=3600 (1 hour)	Proofs are mostly-immutable; pending->upgraded transition warrants short cache

The two contest-page URLs serve identical content for a drawn contest but use different cache strategies: /results/<token>/<slug> participates in HTTP conditional GETs via Rails' stale?(@contest, public: true) (so polling clients can receive 304 Not Modified), while /facebook_page_contests/<slug> always re-renders. Auditors who care about freshness can poll either; auditors who care about minimizing bandwidth should prefer the canonical token-keyed URL.

14.4 The role of each URL

Each public URL exists for a specific reader. The split is deliberate -- verifiers have a curated one-page index, casual visitors see clean results pages, and auditors have direct programmatic access without HTML parsing.

- /facebook_page_contests/<slug> is the URL the operator typically shares (Facebook posts, Instagram bios, follow-up comments under the original contest post). It uses a human-readable slug so it shows up well in link previews. The verification badge is rendered inline as a one-line strip, but no technical artifacts (seed, hash, downloads) are surfaced here. Goal: clean, professional, shareable.
- /results/<token>/<slug> is the canonical token-keyed URL. Identical rendered content; used wherever a token is the primary identifier (the audit-log JSON's proof_url fields, etc.). Same clean strip.
- /results/<token>/verify is the curated verifier index. Shows the badge again with explicit explanation, plus the seed, the SHA-256, links to the eligible-IDs download, audit-log JSON, and latest Bitcoin-anchor proof. Cross-links to the step-by-step guide and this document.
- The three /results/<token>/{eligible-ids.txt,audit-log.json,audit-anchors/*.ots} endpoints are machine-readable surfaces aimed at programmatic verification (Ruby snippet, jq, ots verify).
- /facebook_page_contests/<slug>/audit-log.json is a second mount of the same audit-log JSON, served by the same ContestAuditChainSerializer as the token-keyed twin. It exists for one reason: a verifier who only knows the slug URL (because that's what the operator shared on social media) can reach the audit log by appending one segment, without first opening the results page to recover the token. Eligible-IDs and anchor proofs are still only at the token URL because they are not part of the per-snapshot tamper-detection surface

and do not need a second discovery path.

This separation means a verifier can stop at any layer:
- Glance at the badge -> trust the operator's server
- Click into /results/<token>/verify -> confirm via the published artifacts -> trust your own Ruby
- Recompute the chain hashes -> trust SHA-256 only
- Verify the Bitcoin anchor -> trust the Bitcoin blockchain

15. Threat Model and Defense Matrix

Every honest fairness claim names the threats it defeats and the threats it does not. The matrix below maps specific attack scenarios to the layer of defense that closes them, plus the residual risk in each row.

Scenario	Layer that defeats it	Residual risk
Operator rigs winner via the application UI	No such UI exists. There is no operator-controlled "select winner" button at any point in the dashboard.	None -- there is nothing to bypass.
Operator rigs winner via Rails console (update!, update_columns, update_all)	PostgreSQL BEFORE UPDATE trigger on contest_entries (§7.2) rejects any mutation of winner / disqualification columns once status IN ('drawn', 'completed')	None at this layer -- the trigger fires at the database.
Operator rigs winner via raw SQL	Same trigger as above.	None.
Operator with PG superuser disables the trigger, rewrites winners, restores the trigger	Operation is logged in PostgreSQL's own query log; the audit chain (§8) records no draw event for the rigged result so chain replay reveals no provenance for the new winners. Bitcoin anchor (§9.7) of the original chain head still proves what the contest's chain looked like at the original draw time.	Real, but detectable by anyone holding the audit-log snapshot from before the rewrite, AND structurally bounded by Bitcoin's published commitment to the original chain head.
Operator inserts a forged audit event	before_create :assign_hash_chain computes the canonical hash including all prior chain content; an inserted-by-SQL row with arbitrary content has the wrong entry_hash and the next event's previous_hash points away from it. The verifier (§8.4) raises ChainBroken on the first inconsistency.	None at the cryptographic level. Detection requires running verify_chain! or pulling audit-log.json.
Operator deletes an audit event	BEFORE DELETE trigger refuses unless the documented parent-cascade GUC override is set; even then, the chain's hash pointers indict the deletion.	None for non-cascade deletes; cascade deletes accompany contest destruction and leave no protected event behind.
Operator with PG superuser rewrites both contest_entries AND contest_audit_events and recomputes every downstream hash to maintain chain consistency, all within ~6 hours of the original draw (before Bitcoin confirms)	Application layer cannot defeat this -- it is by definition a database-administrator threat. The pending OpenTimestamps proof would no longer match the rewritten chain head, but during the pending window an attacker could re-submit the rewritten head to OTS instead.	Real but time-bounded to ~6 hours. After Bitcoin confirms the original anchor, the rewrite would also need to rewrite Bitcoin's blockchain.
Operator with PG superuser does the same rewrite >6 hours after the original draw (the Bitcoin block has already confirmed the original anchor)	Bitcoin's confirmed block embeds the digest of the original chain head; any rewrite produces a different chain head and cannot match the Bitcoin-anchored proof. Anyone running ots verify against the saved proof file sees the mismatch immediately.	None at the cryptographic level. The operator can deny service (delete records, refuse to serve the JSON) but cannot forge undetectably.
Operator stops running the anchoring jobs	AdminDailyDigestJob#check_stale_anchors records a SystemAlert(source: "audit_anchor", severity: "warning") for any contest active in the last 7 days whose latest event is >24 hours old without an anchor. An external watchdog polling audit-log.json will also see anchors stop appearing.	Detectable internally within 24 hours; detectable externally by anyone polling the JSON. The operator can stop anchoring future events but cannot retroactively un-anchor past events.

Operator deletes the entire contest record (cascade destruction)	The cascade activates the pick_a_winner.allow_audit_purge GUC override and removes the contest's audit events and chain anchors along with the contest row (§8.3, §12.6). The .ots proof bytes that external parties downloaded before the destruction continue to verify against the Bitcoin blockchain; audit-log.json snapshots taken before the destruction retain the full chain history.	Real -- closeable only by the "External transcript snapshot store" project follow-up. Detectable externally by any watchdog holding a prior audit-log.json snapshot. The operator can erase their own copy but cannot revoke the snapshots others already hold.
Operator manipulates the Graph API response before it enters the database	None -- the API response is not cryptographically signed by Meta. The system can prove what it drew from (every byte goes into draw_eligible_ids and the seeded shuffle) but cannot prove that input matched what was publicly posted.	Real, structural. Defeated only by comparing the eligible-IDs list against the publicly-visible post -- which any contestant can do.
Two concurrent workers both try to draw the same contest	pg_try_advisory_lock(DRAW_WINNERS_NAMESPACE, contest_id); the second worker silently no-ops.	None.
Two concurrent code paths try to write audit events for the same contest at the same instant	pg_advisory_xact_lock(AUDIT_EVENT_CHAIN, contest_id) serializes chain insertions per contest.	None.
A contestant comments 1,000 times to get 1,000 entries	Unique index on (facebook_page_contest_id, facebook_profile_id); upsert_all keeps one entry per profile, latest write wins.	None -- the database enforces one-entry-per-person.
Bitcoin re-orgs after our anchor is confirmed	After ~6 confirmation blocks (~1 hour beyond initial confirmation), re-org probability is statistically negligible.	Practically zero; would invalidate not just our proof but every other application using OpenTimestamps anchoring in the same window.

15.1 The trust ladder, in one sentence each

For a verifier deciding how deep to go:

1. Look at the badge -- trust the operator's server.
2. Re-run the shuffle in Ruby -- trust SHA-256 and your own interpreter.
3. Read the audit log JSON -- same trust as the badge, but you see every event.
4. Recompute the chain hashes yourself -- trust SHA-256 only; the server can no longer hide a chain break.
5. Verify the Bitcoin anchor -- trust Bitcoin's blockchain. This is the strongest single-contest trust level the system offers; it defeats every threat in the matrix above except the named structural ones (legacy contests, pending-anchor window, source-data integrity).
6. Snapshot and diff over time -- trust your own storage. Detects retroactive rewrites even if the chain stays internally consistent.
7. Statistical sanity across many contests -- trust statistics. Detects systematic bias the per-contest cryptographic checks cannot -- e.g., a hypothetical operator who manipulated the input list before hashing it. Requires a sample (typically 30 contests) before tests have meaningful power.

See /verify-fairness for the practical steps at each level. The numbering matches the seven levels on that page.

The whole system, in one line: it's a "we can't backdate or rewrite contest results without you noticing" guarantee. The triggers, hash chain, and Bitcoin anchor do not make the operator trustworthy in some abstract sense -- they make the cost of an undetected rewrite equal to rewriting the Bitcoin blockchain, and the cost of a detected one paid publicly in front of every contestant holding a prior snapshot. That asymmetry, not any single layer, is the fairness guarantee.