Phage
SECURITY

— SECURITY REVIEW · PREPARED FOR

Trepa

DATE

07.05.2026

CONDUCTED BY

PYRO, Security Researcher
YANECA B, Security Researcher
ZUHAIBMOHD, Security Researcher
KYAN, Security Researcher
ISHWAR KUMAR, Security Researcher

COMMIT

7724e94

REMEDIATION

6ad63ee

Table of Contents

About Phage Security

Phage Security is a team of highly skilled smart contract security researchers collaborating with 10+ proven freelancers who have excelled in public contests and private audits alike. With numerous vulnerabilities uncovered and patched across our completed audits, we strive to deliver the absolute best security service and client experience. While 100% security can never be guaranteed, we are committed to giving our utmost effort to protect your protocol.

Check out our previous work at PhageSecurity.com or reach out on X to Pyro.

Disclaimer

Audits are a time, resource, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

System overview

Trepa is a Solana-based short-horizon forecasting game (Flash Pools): each round every player pays the same fixed entry fee (e.g. USDC) and submits a numeric price estimate for an asset (starting with Bitcoin).

After settlement, each player's error (distance to the realized price) is compared to the field median error. Players below the median typically win the round (about half the field in large pools), and those at or above it lose their entry for that round.

Winners get their entry back and share a prize pool funded from losers' entries (after a platform take); among winners, closer estimates earn larger shares of that pool (accuracy-weighted split), subject to product rules such as per-round profit caps in edge cases.

Separately, players earn a precision score and can qualify for streak mechanics; a portion of the take flows to a player accumulator used for streak-related payouts.

Overall Security Posture

The following sections describe the security posture of the Trepa protocol before and after the fix review.

The fix review is the stage of the audit where Phage Security validated the remediations implemented by the Trepa team for the issues reported during the audit.

Before remediation

Trepa is a clean Solana protocol where the on-chain program intentionally stays minimal — stake custody, Merkle-bound payouts, per-winner ROI enforcement — and pushes the heavy logic (reward math, oracles, indexing) into the backend. That works, but it puts most of the protocol's correctness behind one resolver-signed boundary. The chain commits whatever the resolver hands it, so when the backend gets something wrong, the on-chain program has limited ability to catch it.

The findings reflect that pattern:

- **Oracles trusted too generously** (H-01, L-02, M-01) — the external data provider (e.g. Binance) returned a truncated page, Pyth returned partial candles, and the indexer didn't enforce event ordering, but each was treated as ground truth.
- **Settlement math drift** (M-02, M-05, L-06) — WASM misallocated dividends to takeout, used a hardcoded `max_roi` that didn't match the on-chain config, and `protocol_fee` was unbounded at finalize while `platform_fee` was dead state.
- **Claim flow idempotency** (H-02, M-03, M-04) — streak claims weren't bound to a reward ID, settlement wasn't idempotent under RPC failures, and non-qualifying streak users weren't reset until someone else won.
- **On-chain robustness gaps** (M-06, L-03, L-04, L-07) — Token-2022 ATA derivation used legacy SPL form, vault donations could brick finalization, and account sizing used `mem::size_of` instead of `INIT_SPACE`.
- **Auth and signer scoping** (L-01, L-05) — JWT expiry was ~1000× too long due to a ms/sec mismatch, and four privileged signers shared one backend process.

The general direction throughout the review was: lift critical invariants onto the on-chain program, validate oracle and indexer freshness instead of assuming it, and make money-moving flows idempotent.

After remediation

The Trepa team responded promptly and shipped fixes for 14 of the 15 reported issues during the remediation window. The remaining item (L-05) is being tracked as an ongoing infrastructure evaluation.

Most of the patches go beyond the literal bug. The external-data-provider oracle (e.g. Binance) now queries a single nearest-before trade with drift bounds, eliminating the truncation class. Streak claims bind the signed transaction to the reward ID, killing the double-payout primitive and the related idempotency gap. Reward distribution is cap-aware, Pyth coverage is validated, Token-2022 ATA derivation is program-aware, and the `max_roi` and fee-bounding gaps are closed.

The codebase is in good shape for mainnet.

Security Recommendations

- **Set up a bug bounty program** to incentivize ongoing third-party review beyond the audit window.
- **Active invariant monitoring with alerting** — track `total_stake` vs. pool ATA balance, `claims_left` / `open_predictions_left` consistency, and `prediction_revisions` drift between chain and indexer; page on any divergence.
- **Resolver key hygiene** — document signing-service policies, rotation procedure, and incident-response runbook for compromised resolver/creator/admin keys.
- **Anomaly dashboards** — surface resolver fee decisions, max-ROI utilization, unclaimed reward accumulation, and per-pool settlement deviations for community/internal review.
- **Progressive decentralization** — evaluate multi-resolver consensus or on-chain outcome oracles as the protocol matures to reduce single-resolver trust.

Technical Overview

Trepa Predictions Program (Solana / Anchor)

Three privileged roles operate the platform:

- **Admin** — initializes the program, freezes/unfreezes globally, updates platform parameters (treasury, self-claim window, platform fee), and proposes new creator/resolver via a two-step (propose / accept) handover.
- **Creator** — creates pools, registers streaks, sponsors gas for users (acts as the only allowed external `fee_payer`), can claim on behalf of users after the self-claim window expires, and closes pool/prediction accounts.
- **Resolver** — finalizes pools by submitting the Merkle root, the protocol fee, and the winner count; authorizes streak reward payouts. The team is currently evaluating remote signing services to separate resolver key custody from the rest of the backend (see L-05).

Pools snapshot the `platform_fee` from config at creation, so later admin changes do not retroactively affect pools already in flight. Each pool also tracks a running `total_stake` (used as the fee base at finalization) and a `prediction_revisions` counter that the off-chain indexer uses as a sync cursor.

The main user-facing flows are:

- **predict** — stake tokens, lock in a `prediction` value within `[min_outcome, max_outcome]` aligned to `step`, and create a `PredictionAccount` PDA seeded by `(pool, predictor)`.
- **update_prediction / update_stake** — modify the predicted value or top-up the stake (stake-only-up) before the prediction window ends.
- **finalize_pool** — resolver submits `(merkle_root, protocol_fee, claims)`. Enforces `protocol_fee <= total_stake * pool.platform_fee / 10_000`, transfers the streak share (if linked) and the remainder to treasury.
- **claim_rewards** — submit a Merkle proof of `(predictor, amount)` against `pool.root`. Enforces the per-predictor `max_roi` cap, transfers the payout, marks the prediction claimed, and closes the prediction account in-line (refunding rent to either the predictor or the creator depending on who paid for it).
- **close_prediction_account / close_pool_account** — clean up rent after the pool is fully resolved; pool closure also sweeps any residual tokens to the treasury before closing the pool's ATA.
- **claim_streak_rewards** — resolver-gated transfer from a streak accumulator's ATA to a user. Each call creates a `StreakRewardReceipt` PDA seeded by `(reward_id, streak_accumulator, user)` whose `init` constraint makes any second claim with the same `reward_id` revert, preventing replay.

Pool Lifecycle (states)

1. `Active` — `predict`, `update_prediction`, `update_stake` accepted; pool funds accumulate in the pool's ATA and `total_stake` tracks them.
2. `PredictionsFrozen` — predictions blocked, claims still allowed.
3. `ClaimsFrozen` — claims blocked, predictions still allowed.
4. `Frozen` — all operations blocked; admin can still toggle status.
5. **Finalized** (orthogonal to status) — `pool.finalized_timestamp != 0` and `pool.root` set; `claims_left` initialized to the winner count and `total_stake` reduced by the transferred protocol fee. Within `self_claim_window` only the predictor can claim; afterwards the creator may claim on a winner's behalf.

6. **Closed** — all winners claimed (`claims_left == 0`) and all `PredictionAccount`s closed (`open_predictions_left == 0`); residual tokens swept to treasury, pool ATA closed, rent refunded to creator. Legacy `version == 0` pools cannot be closed.

Executive summary

Overview

Project Name	Trepa
Solana Repository	https://github.com/TrepaOrg/trepa-onchain-svm
Solana Commit hash	7724e9400bc0fc8eee170fa008acbc76216e3ba2
Solana Remediation	6ad63ee8f30a4a20e71dac0835680ec8042ed87d
BE Repository	https://github.com/TrepaOrg/trepa-api
BE Commit hash	b1a8ae35ea6a54287284f0bdbd5c0c9addc74d72
BE Remediation	c0c3d3fd468eaccf3a1c960560a343fc34b7deb6
Methods	Manual review

Timeline

Audit kick-off	22.04.2026
End of audit	26.04.2026
Remediations start	27.04.2026
Remediations end	05.05.2026

Scope

trepa-onchain-svm/programs/trepa/src/lib.rs

trepa-onchain-svm/programs/trepa/src/utils.rs

trepa-onchain-svm/programs/trepa/src/state.rs

trepa-onchain-svm/programs/trepa/src/context.rs

trepa-onchain-svm/utils/**

trepa-api/libs/shared/src/services/web3.service.ts

Issues Found

Severity	Count
High	2
Medium	6
Low	7

Findings Summary

2 High · 6 Medium · 7 Low

ID	Title	Severity	Status
[H-01]	Backend external-data-provider oracle can finalize pools with a stale BTCUSDT price from a capped aggregate-trades page	High	Fixed
[H-02]	Streak Reward Claim Transaction Binding Bypass (Double Payout)	High	Fixed
[M-01]	Indexer Consistency & Resolution Trust Can Lead to Incorrect Payout Finalization	Medium	Fixed
[M-02]	WASM reward cap waterfilling underpays uncapped winners	Medium	Fixed
[M-03]	Non-Idempotent Streak Reward Settlement Can Lead to Duplicate Payouts Under Failure Conditions	Medium	Fixed
[M-04]	Non-qualifying streak participants are not reset until someone wins	Medium	Fixed
[M-05]	Pool can be permanently bricked when a winner's leaf exceeds 'max_roi'	Medium	Fixed
[M-06]	Token-2022 ATA derivation mismatch can DoS 'finalize_pool' on time-series pools	Medium	Fixed
[L-01]	Access tokens remain valid for 41 days due to millisecond expiry passed as JWT seconds	Low	Fixed
[L-02]	Pyth volatility calculation accepts partial candle data and can distort streak precision scores	Low	Fixed
[L-03]	Pool vault donations can block the shipped finalization helper before resolver submission	Low	Fixed
[L-04]	Pool finalization can commit unreachable claim counts and permanently deadlock cleanup	Low	Fixed
[L-05]	Multiple privileged operational roles are concentrated behind one backend signing boundary	Low	Acknowledged
[L-06]	'platform_fee' is dead config, while 'protocol_fee' is unbounded at finalization	Low	Fixed
[L-07]	Unsafe account sizing via `std::mem::size\{}_of` in init constraints can break future upgrades	Low	Fixed

Findings

High Severity

[H-01] Backend external-data-provider oracle can finalize pools with a stale BTCUSDT price from a capped aggregate-trades page

Scope note: This is an out-of-scope API/backend finding under the published audit scope.

`BinanceService.getOutcome()` queries the configured external data provider (Binance, in this audit's BTCUSDT pool) to resolve a pool using the last BTCUSDT aggregate trade before the pool reference timestamp. The current implementation queries a 120-second historical range and assumes the response contains the full range.

Buggy code in `trepa-api/libs/shared/src/services/binance.service.ts`:

```
const startTime = Math.floor(outcomeMs - this.LOOKBACK_SECONDS * 1000);

const { data } = await firstValueFrom(
  this.httpService.get<BinanceAggTrade[]>(
    `${this.apiBaseUrl}/api/v3/aggTrades`,
    {
      params: {
        symbol: this.SYMBOL,
        startTime,
        endTime: outcomeMs,
        limit: 1000,
      },
      timeout: 30_000,
    },
  ),
);

const before = data.filter((t) => t.T < outcomeMs);

if (before.length) {
  const last = before[before.length - 1];
  return Number(last.p);
}
```

This is unsafe because the data provider caps `/api/v3/aggTrades` at `limit = 1000`. The provider's API behavior means a `startTime + endTime` query returns the oldest records from the range up to the limit. If the 120-second window contains more than 1000 aggregate trades, Trepa receives only the oldest page and incorrectly treats the last row in that page as the last trade before the reference time.

Official docs:

- `aggTrades` max `limit` is `1000`.
- `startTime + endTime` behaves like a `startTime` query capped by `limit`.
- `endTime`-only queries return the most recent items up to `endTime`.

Sources:

- Provider Spot REST API docs on GitHub, human-readable Markdown view: `https://github.com/binance/binance-spot-api-docs/blob/master/rest-api.md`

- Provider Spot API changelog on GitHub, human-readable Markdown view: https://github.com/binance/binance-spot-api-docs/blob/master/CHANGELOG.md

Illustrative example:

```
reference time = 12:00:00.000
Trepa queries = 11:58:00.000 to 12:00:00.000, limit 1000

Actual trades in range: 1800
Provider returns: oldest 1000 trades
Trepa selects: trade #1000, e.g. 11:59:12
Correct trade: trade #1800, e.g. 11:59:59.900
```

We used the following local PoC test while validating this report:

```
trepa-api/libs/shared/src/services/binance.service.spec.ts
```

This file is not required for the project team to reproduce the issue. It is included here to show the exact mock scenario used during validation.

Key test logic:

```
const fullWindow = Array.from({ length: 1_200 }, (_, index) => {
  const isTrueLastTrade = index === 1_199;
  return makeTrade(
    index + 1,
    startTime + index * 100,
    isTrueLastTrade ? '200.00' : '100.00',
  );
});

const cappedOldestPage = fullWindow.slice(0, 1_000);
const staleReturnedTrade = cappedOldestPage[cappedOldestPage.length - 1];
const trueLastTradeBeforeOutcome = fullWindow[fullWindow.length - 1];

const httpService = {
  get: jest.fn().mockReturnValue(
    of({
      data: cappedOldestPage,
    } as AxiosResponse<BinanceAggTradeFixture[]>),
  ),
} as unknown as HttpService;

const service = new BinanceService(httpService, apiBaseUrl);
const outcome = await service.getOutcome(new Date(outcomeMs));

expect(outcome).toBe(Number(staleReturnedTrade.p));
expect(outcome).toBe(100);
expect(outcome).not.toBe(Number(trueLastTradeBeforeOutcome.p));
```

The test proves Trepa returns the stale capped-page price `100` instead of the true last pre-reference price `200`.

The test output was:

```
PASS libs/shared/src/services/binance.service.spec.ts
  BinanceService
    ✓ POC: capped oldest aggTrades page makes getOutcome return trade #1000
       instead of the true last pre-reference trade
```

The test printed this proof object:

```
{
  "proof": "Binance returned a capped oldest-first page, and getOutcome
     selected the page tail as the outcome.",
  "requestedWindow": {
    "symbol": "BTCUSDT",
    "lookbackMs": 120000,
    "requestedLimit": 1000
  },
  "simulatedReality": {
    "totalTradesInWindow": 1200,
    "trueLastTradeBeforeReference": {
      "id": 1200,
      "lagMsBeforeReference": 100,
      "price": 200
    }
  },
  "binanceResponseSeenByTrepa": {
    "returnedTrades": 1000,
    "selectedReturnedTrade": {
      "id": 1000,
      "lagMsBeforeReference": 20100,
      "price": 100
    }
  },
  "bug": {
    "staleTradeWasSelected": true,
    "missedNewerTradeCount": 200,
    "missedTimeMs": 20000,
    "missedPriceDelta": 100,
    "returnedOutcome": 100,
    "correctOutcomeIfFullWindowWereKnown": 200
  }
}
```

We also used a standalone live probe script during validation:

```
trepa-onchain-svm/pocs_and_reports/pocs_tests/binance_oracle_repro.ts
```

This script is a zero-project-dependency demonstration of the same external-data-provider query behavior. It uses Node's built-in `fetch`; `npx tsx` is only used to run the TypeScript file locally.

Full standalone script:

```
/**
 * Standalone repro + fix demonstrator for the Binance aggTrades truncation
 * bug in `trepa-api/libs/shared/src/services/binance.service.ts`.
 *
 * No dependencies: uses Node 18+ built-in fetch. No @nestjs/axios, no rxjs,
```

```
 * no mocha.
 *
 * Run:
 *   npx tsx binance_oracle_repro.ts
 *
 * Or convert to `.mjs` (strip the type annotations) and run with plain node.
 *
 * Modes (first CLI arg):
 *   probe     (default) - one live query, reports truncation + drift
 *   buggy             - runs the production retrieval against now()
 *   fixedWide          - endTime-only + limit=1000 + chronological last
 *   fixedTight         - endTime-only + limit=1 (recommended production fix)
 *   compare           - runs buggy + both fixes, prints deltas
 */

const BASE_URL = "https://api.binance.us";
const SYMBOL = "BTCUSDT";
const LOOKBACK_SECONDS = 120;
const LIMIT = 1000;
const MAX_ACCEPTABLE_DRIFT_MS = 1_000; // 1 second, for the fixed version

interface BinanceAggTrade {
  a: number; // aggregate trade ID
  p: string; // price (decimal string)
  q: string; // quantity (decimal string)
  f: number; // first trade ID
  l: number; // last trade ID
  T: number; // trade time (ms)
  m: boolean;
  M: boolean;
}

async function fetchAggTrades(
  params: Record<string, string | number>,
): Promise<BinanceAggTrade[]> {
  const url = new URL(`${BASE_URL}/api/v3/aggTrades`);
  for (const [k, v] of Object.entries(params)) {
    url.searchParams.set(k, String(v));
  }
  const res = await fetch(url, { signal: AbortSignal.timeout(30_000) });
  if (!res.ok) {
    throw new Error(`Binance ${res.status} ${res.statusText}`);
  }
  const data = await res.json();
  if (!Array.isArray(data)) {
    throw new Error("Binance returned non-array response");
  }
  return data as BinanceAggTrade[];
}

/**
 * Mirrors the production `getOutcome` in binance.service.ts.
 * This is THE BUG: uses startTime + endTime + limit=1000, takes data[len-1].
 */
async function getOutcome_buggy(timestamp: Date): Promise<number> {
  const outcomeMs = timestamp.getTime();
  const startTime = Math.floor(outcomeMs - LOOKBACK_SECONDS * 1000);

  const data = await fetchAggTrades({
    symbol: SYMBOL,
    startTime,
```

```
    endTime: outcomeMs,
    limit: LIMIT,
  });

  const before = data.filter((t) => t.T < outcomeMs);
  if (!before.length) {
    throw new Error("No trades before outcomeMs");
  }
  const last = before[before.length - 1];
  return Number(last.p);
}

/**
 * Fix variant A (wide): drop startTime so Binance returns the 1000 newest
 * trades at or before endTime. Response is still chronologically ordered
 * (oldest-to-newest), so the last element is the newest. Still hits the
 * 1000-row cap because the cap is inherent to the endpoint, but now you get
 * the right 1000 rows. Useful for diagnostics; less defensible for production.
 */
async function getOutcome_fixedWide(
  timestamp: Date,
  maxDriftMs: number = MAX_ACCEPTABLE_DRIFT_MS,
): Promise<number> {
  const outcomeMs = timestamp.getTime();

  const data = await fetchAggTrades({
    symbol: SYMBOL,
    endTime: outcomeMs,
    limit: LIMIT,
  });

  const before = data.filter((t) => t.T < outcomeMs);
  if (!before.length) {
    throw new Error("No trades before outcomeMs (fixedWide path)");
  }

  const last = before[before.length - 1];
  const driftMs = outcomeMs - last.T;
  if (driftMs > maxDriftMs) {
    throw new Error(
      `fixedWide path rejected: nearest-before trade is ${driftMs}ms stale ` +
        `(max ${maxDriftMs}ms)`,
    );
  }
  return Number(last.p);
}

/**
 * Fix variant B (tight, recommended): ask Binance directly for the single
 * aggregate trade at or before `outcomeMs - 1`. Strict-before semantic is
 * encoded in the query itself, not in a post-fetch filter. No truncation
 * exposure, no ordering ambiguity, and no reliance on response ordering.
 */
async function getOutcome_fixedTight(
  timestamp: Date,
  maxDriftMs: number = MAX_ACCEPTABLE_DRIFT_MS,
): Promise<number> {
  const outcomeMs = timestamp.getTime();

  const data = await fetchAggTrades({
    symbol: SYMBOL,
```

```
    endTime: outcomeMs - 1, // inclusive endTime; -1 gives strict-before
    limit: 1,
  });

  const trade = data[0];
  if (!trade || trade.T >= outcomeMs) {
    throw new Error(
      "No valid Binance trade strictly before reference timestamp",
    );
  }

  const driftMs = outcomeMs - trade.T;
  if (driftMs > maxDriftMs) {
    throw new Error(
      `fixedTight path rejected: nearest-before trade is ${driftMs}ms stale ` +
        `(max ${maxDriftMs}ms)`,
    );
  }
  return Number(trade.p);
}

async function probe(): Promise<void> {
  const outcomeMs = Date.now();
  const startTime = outcomeMs - LOOKBACK_SECONDS * 1000;

  const [data, nearestData] = await Promise.all([
    fetchAggTrades({
      symbol: SYMBOL,
      startTime,
      endTime: outcomeMs,
      limit: LIMIT,
    }),
    fetchAggTrades({
      symbol: SYMBOL,
      endTime: outcomeMs - 1,
      limit: 1,
    }),
  ]);

  const first = data[0];
  const lastReturned = data[data.length - 1];
  const nearest = nearestData[0];
  const hitLimit = data.length === LIMIT;
  const confirmedMissedNewerTrade =
    nearest !== undefined && nearest.T > lastReturned.T;
  const windowSpannedMs = lastReturned.T - first.T;
  const staleSelectionLagMs = outcomeMs - lastReturned.T;
  const nearestTradeLagMs = nearest ? outcomeMs - nearest.T : null;
  const missedTimeMs = nearest ? nearest.T - lastReturned.T : null;
  const priceDeltaToNearest = nearest
    ? Number(nearest.p) - Number(lastReturned.p)
    : null;

  const report = {
    symbol: SYMBOL,
    requestedWindowMs: LOOKBACK_SECONDS * 1000,
    returnedRowCount: data.length,
    hitLimit,
    confirmedMissedNewerTrade,
    firstReturnedT: new Date(first.T).toISOString(),
    lastReturnedT: new Date(lastReturned.T).toISOString(),
```

```
    requestedEndTime: new Date(outcomeMs).toISOString(),
    windowSpannedMs,
    staleSelection: {
      tradeId: lastReturned.a,
      tradeTime: new Date(lastReturned.T).toISOString(),
      lagMs: staleSelectionLagMs,
      price: Number(lastReturned.p),
    },
    nearestStrictlyBeforeReference: nearest
      ? {
          tradeId: nearest.a,
          tradeTime: new Date(nearest.T).toISOString(),
          lagMs: nearestTradeLagMs,
          price: Number(nearest.p),
        }
      : null,
    missedTimeMs,
    priceDeltaToNearest,
    verdict: confirmedMissedNewerTrade
      ? `CONFIRMED STALE: production-shaped query selected trade ${
          lastReturned.a}, but endTime-only limit=1 found newer trade ${
          nearest!.a} ${missedTimeMs} ms later`
      : hitLimit
        ? `HIT LIMIT: response returned exactly ${LIMIT} rows,
           but this sample did not confirm a newer missed trade`
        : `NOT HIT LIMIT: production-shaped query returned ${data.length} rows`,
  };

  console.log(JSON.stringify(report, null, 2));
}

async function compare(): Promise<void> {
  const t = new Date();
  const [buggy, wide, tight] = await Promise.all([
    getOutcome_buggy(t).catch((e: Error) => `ERROR: ${e.message}`),
    getOutcome_fixedWide(t).catch((e: Error) => `ERROR: ${e.message}`),
    getOutcome_fixedTight(t).catch((e: Error) => `ERROR: ${e.message}`),
  ]);
  const asNum = (x: unknown) => (typeof x === "number" ? x : null);
  console.log(
    JSON.stringify(
      {
        referenceTime: t.toISOString(),
        buggy,
        fixedWide: wide,
        fixedTight: tight,
        buggyMinusTight:
          asNum(buggy) !== null && asNum(tight) !== null
            ? (asNum(buggy) as number) - (asNum(tight) as number)
            : null,
        wideMinusTight:
          asNum(wide) !== null && asNum(tight) !== null
            ? (asNum(wide) as number) - (asNum(tight) as number)
            : null,
      },
      null,
      2,
    ),
  );
}
```

```
async function main(): Promise<void> {
  const mode = process.argv[2] ?? "probe";
  switch (mode) {
    case "probe":
      await probe();
      break;
    case "buggy":
      console.log(await getOutcome_buggy(new Date()));
      break;
    case "fixedWide":
      console.log(await getOutcome_fixedWide(new Date()));
      break;
    case "fixedTight":
      console.log(await getOutcome_fixedTight(new Date()));
      break;
    case "compare":
      await compare();
      break;
    default:
      console.error(`Unknown mode: ${mode}`);
      process.exit(1);
  }
}

main().catch((err) => {
  console.error(err);
  process.exit(1);
});
```

Example commands and outputs:

```
npx --yes tsx binance_oracle_repro.ts probe
```

Output:


```
{
  "symbol": "BTCUSDT",
  "requestedWindowMs": 120000,
  "returnedRowCount": 1000,
  "hitLimit": true,
  "confirmedMissedNewerTrade": true,
  "firstReturnedT": "2026-04-24T13:53:20.852Z",
  "lastReturnedT": "2026-04-24T13:54:46.606Z",
  "requestedEndTime": "2026-04-24T13:55:20.697Z",
  "windowSpannedMs": 85754,
  "staleSelection": {
    "tradeId": 3944251517,
    "tradeTime": "2026-04-24T13:54:46.606Z",
    "lagMs": 34091,
    "price": 78016.22
  },
  "nearestStrictlyBeforeReference": {
    "tradeId": 3944252401,
    "tradeTime": "2026-04-24T13:55:20.545Z",
    "lagMs": 152,
    "price": 77957.17
  },
  "missedTimeMs": 33939,
```

```
  "priceDeltaToNearest": -59.05000000000291,
  "verdict": "CONFIRMED STALE: production-shaped query selected trade
     3944251517, but endTime-only limit=1 found newer trade 3944252401 33939
     ms later"
}
```


```
npx --yes tsx binance_oracle_repro.ts buggy
```

Output:

```
78002.28
```

```
npx --yes tsx binance_oracle_repro.ts fixedWide
```

Output:

```
78000
```

```
npx --yes tsx binance_oracle_repro.ts fixedTight
```

Output:

```
78011.14
```

```
npx --yes tsx binance_oracle_repro.ts compare
```

Output:


```
{
  "referenceTime": "2026-04-24T13:55:06.199Z",
  "buggy": 78001.71,
  "fixedWide": 78011.14,
  "fixedTight": 78011.14,
  "buggyMinusTight": -9.429999999993015,
  "wideMinusTight": 0
}
```


Note: `buggy`, `fixedWide`, and `fixedTight` are live market queries. Individual command outputs may differ if run at different moments. The `compare` mode is the cleanest direct comparison because it runs all three retrieval strategies against the same reference timestamp.

Impact:

A stale provider-supplied outcome feeds directly into pool resolution. The stale value is rounded, queued as `RESOLVE_POOL`, used by `pools-resolution.service.ts` to calculate winners/rewards,

stored in the `resolutions` table, and committed on-chain through the resolver-signed `finalizePool` transaction. This can produce wrong winners, wrong losers, wrong reward amounts, and a valid on-chain Merkle root for an economically incorrect outcome.

Likelihood:

High. BTCUSDT frequently has high aggregate-trade volume, and live testing showed the 1000-row cap can be hit during normal operation. The failure is silent: a capped response is still a successful HTTP response, so the retry logic does not trigger and no truncation warning is emitted.

Severity rationale:

If considered in scope, this should be High because it can deterministically corrupt the economic outcome of an entire pool. The bug does not require provider manipulation or chain-level compromise; it is triggered whenever the queried 120-second BTCUSDT window contains more than 1000 aggregate trades and the selected capped-page tail differs from the true nearest pre-reference trade. The on-chain program then faithfully commits the resolver-provided Merkle root, so the on-chain proof layer cannot distinguish the stale oracle value from a correct one.

RECOMMENDATION

Replace the wide historical query with a direct nearest-before query. Since the provider's `endTime` is inclusive and Trepa wants `T < outcomeMs`, query with `endTime: outcomeMs - 1` and `limit: 1`.

Why `endTime: outcomeMs - 1` and `limit: 1` is correct:

- The provider's `endTime` is inclusive, so `endTime: outcomeMs` may legally return a trade whose `T === outcomeMs`.
- Trepa's current logic explicitly wants `T < outcomeMs`, not `T <= outcomeMs`.
- Using `endTime: outcomeMs - 1` asks the provider for the newest aggregate trade at or before the millisecond immediately before the reference timestamp.
- Using `limit: 1` asks for exactly one trade, so response ordering and 1000-row page truncation no longer matter.
- The code still validates `trade.T < outcomeMs`, so if the provider ever returns an unexpected boundary trade, the resolver fails closed.
- The actual value of `MAX_ACCEPTABLE_DRIFT_MS` should be chosen and documented by the product/team. `1_000` ms below means 1 second and is a suggested starting point, not a protocol constant.

Replace this buggy block

```
const startTime = Math.floor(outcomeMs - this.LOOKBACK_SECONDS * 1000);

let tradeTime: number | undefined;

const outcome = await this.withRetry(async () => {
  const { data } = await firstValueFrom(
    this.httpService.get<BinanceAggTrade[]>(
      `${this.apiBaseUrl}/api/v3/aggTrades`,
      {
        params: {
          symbol: this.SYMBOL,
          startTime,
          endTime: outcomeMs,
          limit: 1000,
```

```
      },
      timeout: 30_000,
    },
  ),
);

if (!Array.isArray(data)) {
  throw new InternalServerErrorException(
    'Binance aggTrades API returned invalid response',
  );
}

const before = data.filter((t) => t.T < outcomeMs);

if (before.length) {
  const last = before[before.length - 1];
  tradeTime = last.T;
  const price = Number(last.p);
  if (!Number.isFinite(price)) {
    throw new InternalServerErrorException(
      `Binance trade price is invalid for ${this.SYMBOL}`,
    );
  }
  return price;
}

if (data.length) {
  throw new InternalServerErrorException(
    `Binance returned no trades before timestamp for ${this.SYMBOL}: all ${
      data.length} trade(s) in the ${this.LOOKBACK_SECONDS}s window are at
      or after the outcome timestamp`,
  );
}

throw new BinanceTransientError('No trades in window');
});
```

With this bug-free operationally safer code

Add this constant next to the existing service constants:

```
private readonly MAX_ACCEPTABLE_DRIFT_MS = 1_000; // 1 second; choose/document
   product threshold
```

Then replace the block above with:

```
let tradeTime: number | undefined;
let driftMs: number | undefined;

const outcome = await this.withRetry(async () => {
  const { data } = await firstValueFrom(
    this.httpService.get<BinanceAggTrade[]>(
      `${this.apiBaseUrl}/api/v3/aggTrades`,
      {
        params: {
          symbol: this.SYMBOL,
          endTime: outcomeMs - 1,
          limit: 1,
```

```
      },
      timeout: 30_000,
    },
   ),
  );

  if (!Array.isArray(data)) {
   throw new InternalServerErrorException(
    'Binance aggTrades API returned invalid response',
   );
  }

  const trade = data[0];

  if (!trade || trade.T >= outcomeMs) {
   throw new BinanceTransientError(
    `No Binance trade strictly before timestamp for ${this.SYMBOL}`,
   );
  }

  tradeTime = trade.T;
  driftMs = outcomeMs - trade.T;

  if (driftMs > this.MAX_ACCEPTABLE_DRIFT_MS) {
   throw new InternalServerErrorException(
    `Binance nearest trade is too stale for ${this.SYMBOL}: ` +
     `drift=${driftMs}ms max=${this.MAX_ACCEPTABLE_DRIFT_MS}ms`,
   );
  }

  const price = Number(trade.p);

  if (!Number.isFinite(price)) {
   throw new InternalServerErrorException(
    `Binance trade price is invalid for ${this.SYMBOL}`,
   );
  }

  return price;
});
```

Also replace the current debug log:

```
this.logger.debug(
  `Binance ${this.SYMBOL} rawOutcome=${outcome}` +
   `tradeTime=${tradeTime} ` +
   `outcomeTimestamp=${timestampSeconds} startTime=${startTime}`,
);
```

With:

```
this.logger.log(
  `Binance ${this.SYMBOL} rawOutcome=${outcome} ` +
   `tradeTime=${tradeTime} outcomeTimestamp=${timestampSeconds} ` +
   `driftMs=${driftMs}`,
);
```

This fixes the issue because Trepa no longer fetches an incomplete historical page. It asks the

provider for exactly one trade: the nearest aggregate trade at or before `outcomeMs - 1`, which is strictly before the reference timestamp.

Also add regression coverage ensuring:

- The old `startTime + endTime + limit=1000` shape is not used.
- The request uses `endTime: outcomeMs - 1` and `limit: 1`.
- The service rejects trades with `T >= outcomeMs`.
- The service rejects unexpectedly stale trades where `outcomeMs - trade.T > MAX_ACCEPTABLE_DRIFT_MS`.

TEAM

Fixed in trepa-api PR #284. The external-data-provider query (Binance, in this pool) was switched to a direct nearest-before lookup using `endTime: outcomeMs - 1` and `limit: 1`, eliminating the page-truncation window so a stale boundary trade can no longer slip through.

[H-02] Streak Reward Claim Transaction Binding Bypass (Double Payout)

The streak reward claim flow contains a **transaction-binding flaw** where the backend does not verify that the submitted `streak_reward_id` corresponds to the signed transaction.

This allows an attacker to:

- submit a transaction signed for **Reward A**
- but claim it as **Reward B**
- resulting in **incorrect DB state with a valid on-chain payout**
- and enabling **repeatable double-claim / streak pool draining**

Root Cause

The claim flow is split into two steps:

1. `POST /transactions/claim-streak-reward`
   -> builds an unsigned transaction for a specific reward
2. `POST /transactions/claim-streak-reward/submit`
   -> accepts:
   - `streak_reward_id`
   - `signed_transaction`
   - `proof`

Issue

During `/transactions/claim-streak-reward/submit`, the backend:

- verifies signature/proof
- trusts `streak_reward_id` from request
- **does NOT validate that the signed transaction corresponds to that reward**

There is **no binding between**:

- signed transaction payload

- reward ID (DB row)
- claim state

Affected Area (Code Reference)

Relevant flow:

```
create tx from DB row A: transactions.service.ts:702-757
submit marks whichever row was submitted: transactions.service.ts:785-833
proof only checks message+wallet: transaction-verification.service.ts:23-42
```

Problem Pattern

- Transaction is constructed using DB reward row
- But during `/submit`:
  - `signed_transaction` is **not decoded**
  - No validation is performed between:
    - transaction amount
    - streak_id
    - wallet
  - and the provided `streak_reward_id`

Steps to Reproduce

Initial State

Two unclaimed rewards for the same streak:

```
Reward A: $50.15 -> ef8a175f-c2f2-4e79-9b89-a203906ee27e
Reward B: $25.05 -> bef2867e-f24c-40ab-a51f-7c095dcf86b5
```

Step 1 — Create transaction for Reward A

```
POST /transactions/claim-streak-reward
{
  "streak_reward_id": "ef8a175f-c2f2-4e79-9b89-a203906ee27e"
}
```

Step 2 — Intercept submit request

```
POST /transactions/claim-streak-reward/submit
```

Original request:

```
{
  "streak_reward_id": "ef8a175f-c2f2-4e79-9b89-a203906ee27e",
  "signed_transaction": "<tx-for-A>",
  "proof": "<proof-for-A>"
}
```

Step 3 — Tamper request

```
{
  "streak_reward_id": "bef2867e-f24c-40ab-a51f-7c095dcf86b5",  // modified
  "signed_transaction": "<tx-for-A>",                  // unchanged
  "proof": "<proof-for-A>"                           // unchanged
}
```

Step 4 — Submit

- backend accepts request
- DB marks **Reward B as claimed**
- blockchain executes transaction for **Reward A ($50.15)**

Step 5 — Exploit

- Reward A remains unclaimed in DB
- attacker claims Reward A again -> second payout

TEAM

Fixed in trepa-api PR #268. The signed proof is now bound to a context object containing `streak_reward_id` and `wallet_address`; the submit endpoint re-verifies that context before updating `is_claimed`, so swapping the `streak_reward_id` after signing no longer passes verification.

Medium Severity

[M-01] Indexer Consistency & Resolution Trust Can Lead to Incorrect Payout Finalization

The system relies on **off-chain indexed data (DB)** for:

- prediction values
- winner calculation
- reward distribution

However:

1. The indexer may produce **stale or out-of-order data**
2. The resolution logic **trusts this data without verifying completeness**

This combination can lead to:

```
Incorrect indexed data -> incorrect winner calculation -> permanently incorrect
   payouts on-chain
```

Root Cause

1. Indexer does not enforce event ordering

`apps/indexer/src/common/store/indexer-store.provider.ts`

Current behavior:

```
update(prediction, event)
```

No validation of:

- event recency
- ordering
- overwrite safety

This allows:

```
newer value -> overwritten by older event
```

2. Startup race condition

`apps/indexer/src/indexer/indexer.service.ts`

Current pattern:

```
subscribeLiveEvents()
startBackfill()
```

Both run in parallel

This creates a race where:

```
live event (new) -> processed
backfill event (old) -> processed later -> overwrites new
```

3. Resolution relies on weak sync signal

`libs/shared/src/services/web3.service.ts` (validateIndexerSync)

Current check:

```
abs(chain_slot - indexer_slot) < threshold
```

This only checks **proximity**, not:

- whether all events are processed
- whether data is complete
- whether any events were missed

TEAM

Fixed in trepa-onchain-svm PR #141 and trepa-api PR #291. A `prediction_revisions` counter was added to `PoolAccount` (incremented on every `update_stake`/`update_predict` and emitted in events), giving the indexer a deterministic ordering signal independent of slot/timestamp. Resolution now blocks until the indexed revision matches the on-chain value, and the prediction-row count is validated against `open_predictions_left` before finalization.

[M-02] WASM reward cap waterfilling underpays uncapped winners

The WASM reward calculator can route winner-dividend funds into `takeout` while at least one winner is still below their gain cap.

Trepa first determines winners by prediction accuracy, then computes:

```
sum_loser_stakes = total_stake - sum_winner_stakes
takeout = sum_loser_stakes * 20%
dividends = sum_loser_stakes - takeout
```

Those `dividends` are the loser-stake funds intended to be distributed among winners, subject to each winner's max gain cap. If one winner reaches their cap but another winner is still below their cap, the capped winner's excess allocation should be redistributed to the still-uncapped winner. Instead, the current code can leave the uncapped winner underpaid and add the residual to `takeout`.

This is a Medium issue because it is in the production settlement path and can cause partial user underpayment plus protocol/takeout over-allocation during normal pool resolution.

This finding is distinct from the already-submitted `max_roi` drift issue where the WASM calculator uses a hardcoded `100x` cap while the on-chain pool stores a configurable `max_roi`. That issue is about producing leaves that can exceed the pool's on-chain cap and later revert with `MaxRoiExceeded`. This finding assumes the WASM cap being used is the intended cap for the calculation, and shows a separate distribution bug inside the capped-waterfill algorithm.

The production path uses this WASM calculator during resolution:

```
// trepa-api/libs/shared/src/services/pools-resolution.service.ts:20-58
import { RewardsMechanismService } from '@/libs/wasm';

constructor(
  ...
  private readonly rewardsMechanismService: RewardsMechanismService,
  ...
) {}
```

```
// trepa-api/libs/shared/src/services/pools-resolution.service.ts:262-272
const result = await this.rewardsMechanismService.process(
  predictions.map((p) => ({
    address: p.predictor_account,
    stake: p.stake,
    value: p.prediction,
  })),
  pool.min_outcome,
  pool.max_outcome,
  outcomeValue,
```

```
  stdLogReturnValue,
);
```

`RewardsMechanismService` calls the generated WASM package:

```
// trepa-api/libs/wasm/src/rewards-mechanism/rewards-mechanism.service.ts:21-35
const result = JSON.parse(
  this.trepaRewardsMechanismModule.process(
    JSON.stringify(
      predictions.map((p: TrepaPredictionDto) => ({
        address: p.address,
        value: p.value,
        stake: p.stake,
      })),
    ),
    minOutcome,
    maxOutcome,
    outcome,
    stdLogReturns,
  ),
);
```

The module is loaded from the generated `wasm-pack` output:

```
// trepa-api/libs/wasm/src/index.ts:3-10
const wasmModule = require(
  path.join(process.cwd(), 'libs/wasm/pkg/trepa_rewards_mechanism_program.js'),
) as typeof TrepaRewardsMechanismWasm;
```

That generated package is built from `trepa-api/libs/wasm/rust`, whose exported WASM entrypoint calls `TrepaRewardsMechanismProgram::process`:

```
// trepa-api/libs/wasm/rust/src/wasm.rs:6-24
#[wasm_bindgen]
pub fn process(
    predictions_json: &str,
    min_outcome: f64,
    max_outcome: f64,
    outcome: f64,
    std_log_returns: f64,
) -> Result<String, JsValue> {
    let predictions: Vec<TrepaPrediction> = serde_json::from_str(
        predictions_json)
        .map_err(|e| JsValue::from_str(&format!("Failed to parse predictions: {
            }", e)))?;

    let calculator = TrepaRewardsMechanismProgram::new();
    match calculator.process(
        &predictions,
        min_outcome,
        max_outcome,
        outcome,
        std_log_returns,
    ) {
        ...
    }
```

```
}
```

The root cause is in `get_final_alpha()`:

```
// trepa-api/libs/wasm/rust/src/lib.rs:474-520
let dividend_to_weight_ratio = dividends / total_winning_weight;
let cap_to_weight_ratios_values: Vec<Decimal> =
   cap_to_weight_ratios.values().cloned().collect();
let gain_caps: Vec<Decimal> = winner_gains_capped.values().cloned().collect();
let sum_gain_cap: Decimal = gain_caps.iter().sum();

let has_valid_cap_to_weight_ratio = cap_to_weight_ratios_values
   .iter()
   .any(|ratio| dividend_to_weight_ratio <= *ratio);

let final_alpha = if has_valid_cap_to_weight_ratio {
   dividend_to_weight_ratio
} else if dividends >= sum_gain_cap {
   ...
} else {
   ...
};
```

This condition is too weak. It uses raw proportional alpha if **any** winner can receive that alpha without hitting their cap. Raw alpha is only safe if **all** winners can receive that alpha without hitting their cap.

When raw alpha caps one winner but not another, the function should continue into the waterfilling path and recompute alpha for the remaining uncapped winners.

Concrete example:

```
2 winners stake $1 each
190 losers stake $1 each
outcome = 1000
Winner A prediction = 1000, error = 0
Winner B prediction = 901, error = 99
Each loser prediction = 900, error = 100
```

The median error is `100`, so both Winner A and Winner B are winners because their errors are strictly below the median:

```
Winner A: error 0   -> winner
Winner B: error 99  -> winner
Losers:   error 100 -> loser
```

Loser stake is `$190`, so:

```
intended takeout = $190 * 20% = $38
winner dividends = $190 - $38 = $152
```

Reward weights are accuracy-based:

```
error = abs(prediction - outcome)
r = error / median_error
weight = (1 / (1 + r))^6
```

For this example:

```
Winner A error = 0
Winner A weight = (1 / (1 + 0))^6 = 1

Winner B error = 99
Winner B weight = (1 / (1 + 99 / 100))^6
           ~= 0.016102063742887196

total_winning_weight ~= 1.0161020637428873
raw_alpha = dividends / total_winning_weight
       = 152 / 1.0161020637428873
       ~= 149.5912718060002
```

Using that raw alpha:

```
Winner A raw gain = 149.5912718060002 * 1
             = 149.5912718060002
             -> capped to 100

Winner B raw gain = 149.5912718060002 * 0.016102063742887196
             ~= 2.40872819399978
```

After Winner A caps, there are still:

```
remaining dividends = 152 - 100 - 2.40872819399978
              ~= 49.59127180600022
```

Those remaining dividends should be redistributed to Winner B because Winner B is still below their own `100x` gain cap.

Correct cap-aware distribution:

```
Winner A payout = $101
Winner B payout = $53
takeout = $38
```

Current output:

```
Winner A payout = $101
Winner B payout = ~$3.408728
takeout = ~$87.591272
```

So about `$49.591272` that should still be distributed to an uncapped winner is instead added to takeout.

The backend then commits the WASM output into the database and on-chain settlement:

```
// trepa-api/libs/shared/src/services/pools-resolution.service.ts:338-367
const rewards = predictions.map(({ id, predictor_account }) => {
  const winner = result.winners.find(
    (r) => r.address === predictor_account,
  );
  const loser = result.losers.find(
    (r) => r.address === predictor_account,
  );
  const reward = winner?.return ?? 0;
  const amount = roundDown(reward, 6);
  const precision_score =
    winner?.precision_score ?? loser?.precision_score ?? 0;
  return {
    prediction_id: id,
    amount,
    precision_score,
  };
});

const returns = result.winners.map((r) => r.return);

const [resolution] = await tx
  .insert(permissionedSchema.resolutionsTable)
  .values({
    pool_id: id,
    result: outcomeValue,
    median: result.median,
    protocol_fee: result.takeout,
  })
  .returning();
```

```
// trepa-api/libs/shared/src/services/pools-resolution.service.ts:468-478
await this.web3Service.resolvePool(
  id,
  returns,
  adresses,
  result.takeout,
  pool.streak_id,
);
```

```
// trepa-api/libs/shared/src/services/web3.service.ts:315-343
const winners = predictors.map((predictor, index) => {
  return {
    address: predictor,
    prize: scaleToInteger(prizes[index], this.usdcDecimals),
  };
});

const tx = streakId
  ? await finalizeStreakPool(
      this.program,
      this.resolverKeypair.publicKey,
      poolId,
      winners,
      new BN(scaleToInteger(protocolFee, this.usdcDecimals)),
      ...
    )
```

```
: await finalizePool(
    this.program,
    this.resolverKeypair.publicKey,
    poolId,
    winners,
    new BN(scaleToInteger(protocolFee, this.usdcDecimals)),
    ...
  );
```

Proof of concept:

Add the following test to `trepa-api/libs/wasm/rust/src/tests.rs`:

```
#[test]
fn test_partial_cap_leftover_is_misdirected_to_takeout() {
    let program = TrepaRewardsMechanismProgram::new();

    let mut predictions = vec![
        TrepaPrediction {
            address: "exact-winner".to_string(),
            value: 1000.0,
            stake: 1.0,
        },
        TrepaPrediction {
            address: "uncapped-winner".to_string(),
            value: 901.0,
            stake: 1.0,
        },
    ];

    for i in 0..190 {
        predictions.push(TrepaPrediction {
            address: format!("loser-{i}"),
            value: 900.0,
            stake: 1.0,
        });
    }

    let result = program
        .process(&predictions, 0.0, 2000.0, 1000.0, 0.01)
        .unwrap()
        .unwrap();

    let exact_winner = result
        .winners
        .iter()
        .find(|winner| winner.address == "exact-winner")
        .unwrap();
    let uncapped_winner = result
        .winners
        .iter()
        .find(|winner| winner.address == "uncapped-winner")
        .unwrap();

    assert_eq!(exact_winner.r#return, 101.0);
    assert!(
        (uncapped_winner.r#return - 3.408728193999781).abs() < 1e-9,
        "current payout was {}",
        uncapped_winner.r#return
    );
```

```
    assert!(
        (result.takeout - 87.59127180600022).abs() < 1e-9,
        "current takeout was {}",
        result.takeout
    );

    // Correct cap-aware redistribution would keep takeout at the configured
    // 20% of loser stakes and route the remaining winner dividends to the
    // still-uncapped winner.
    assert!(uncapped_winner.r#return < 53.0);
    assert!(result.takeout > 38.0);
}
```

Observed result from the test case:

```
exact-winner payout:    101.0
uncapped-winner payout:  3.408728193999781
takeout:            87.59127180600022

expected exact-winner payout:    101.0
expected uncapped-winner payout:  53.0
expected takeout:            38.0
```

Focused test output:

```
Finished `test` profile [unoptimized + debuginfo] target(s) in 0.10s
Running unittests src/lib.rs (target/debug/deps/trepa_rewards_mechanism_program-
    6f571db8f0b265b1)

running 1 test
test tests::test_partial_cap_leftover_is_misdirected_to_takeout ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 30 filtered out;
    finished in 0.01s
```

Full test-suite output after adding the PoC:

```
running 31 tests
...
test tests::test_partial_cap_leftover_is_misdirected_to_takeout ... ok

test result: ok. 31 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
    finished in 0.01s
```

RECOMMENDATION

Replace the raw-alpha shortcut in `trepa-api/libs/wasm/rust/src/lib.rs:get_final_alpha`.

Current code:

```
let has_valid_cap_to_weight_ratio = cap_to_weight_ratios_values
    .iter()
    .any(|ratio| dividend_to_weight_ratio <= *ratio);

let final_alpha = if has_valid_cap_to_weight_ratio {
```

```
    dividend_to_weight_ratio
} else if dividends >= sum_gain_cap {
    cap_to_weight_ratios_values
        .iter()
        .max()
        .ok_or(CustomError::EmptyCollectionInMathOperation)?
        .clone()
} else {
    // First valid alpha in 0-order (theta is function of error via cap/weight)
    let first_valid_alpha = winners_sorted_by_theta
        .iter()
        .find_map(|(address, _)| {
            is_valid.get(address).and_then(|&v| {
                if v {
                    alpha_candidate.get(address).copied()
                } else {
                    None
                }
            })
        })
        .unwrap_or(Decimal::ZERO);
    first_valid_alpha
};
```

Replace it with:

```
// Raw proportional alpha is only valid when every winner can receive
// alpha * weight without hitting their gain cap. If even one winner caps,
// the capped winner must be removed from the active weight set and alpha
// must be recomputed for the remaining uncapped winners.
let raw_alpha_fits_every_winner_cap = cap_to_weight_ratios_values
    .iter()
    .all(|ratio| dividend_to_weight_ratio <= *ratio);

let final_alpha = if raw_alpha_fits_every_winner_cap {
    dividend_to_weight_ratio
} else if dividends >= sum_gain_cap {
    cap_to_weight_ratios_values
        .iter()
        .max()
        .ok_or(CustomError::EmptyCollectionInMathOperation)?
        .clone()
} else {
    // First valid alpha in 0-order after capped winners are removed from
    // the active weight set.
    winners_sorted_by_theta
        .iter()
        .find_map(|(address, _)| {
            is_valid.get(address).and_then(|&v| {
                if v {
                    alpha_candidate.get(address).copied()
                } else {
                    None
                }
            })
        })
        .ok_or(CustomError::EmptyCollectionInMathOperation)?
};
```

Why this fixes the issue:

- `all()` preserves the raw proportional path only when no winner is capped.
- If one winner caps while another winner remains uncapped, the function falls through to the existing `alpha_candidate` / `is_valid` waterfilling path.
- That path recomputes alpha after cumulative capped gains are removed, so remaining dividends continue flowing to uncapped winners.
- Residual value should only be added to `takeout` after all winners have reached their gain caps.

Also add the PoC test above as a regression test. After the fix, the test expectation should be updated to the corrected distribution:

```
assert_eq!(exact_winner.r#return, 101.0);
assert!(
   (uncapped_winner.r#return - 53.0).abs() < 1e-9,
   "fixed payout was {}",
   uncapped_winner.r#return
);
assert!(
   (result.takeout - 38.0).abs() < 1e-9,
   "fixed takeout was {}",
   result.takeout
);
```

TEAM

Fixed in trepa-api PR #283. The WASM waterfilling algorithm now correctly redistributes capped winners' excess to still-uncapped winners instead of routing it to `takeout`, and explicitly fails on invalid/no-alpha inputs rather than silently falling back to zero.

[M-03] Non-Idempotent Streak Reward Settlement Can Lead to Duplicate Payouts Under Failure Conditions

The streak reward claim flow is **not idempotent**, meaning a single logical claim can result in **multiple on-chain payouts** if the backend fails after transaction broadcast but before persisting the claimed state.

While not trivially exploitable via client-side manipulation alone, this represents a **system-level consistency flaw** that can lead to duplicate payouts under real-world failure conditions (e.g. RPC timeouts, network instability, partial failures).

Root Cause

The current claim flow follows this pattern:

```
1. Verify proof + request

2. Mark reward as claimed (DB transaction context)

3. Send transaction to chain

4. Wait for confirmation

5. If confirmation fails -> rollback DB transaction
```

Relevant code:

```
libs/shared/src/services/transactions.service.ts:785-834
```

The design assumes that transaction confirmation is reliable and tightly coupled with database state — which is not guaranteed.

A realistic failure sequence:

1. User submits claim
2. Backend broadcasts transaction to Solana
3. Transaction succeeds on-chain
4. Backend confirmation fails (timeout / RPC error)
5. DB transaction rolls back -> reward remains unclaimed
6. User retries claim -> second payout occurs

RECOMMENDATION

1. Make Settlement Idempotent (Critical)

```
if (tx may have been broadcast) {
   query chain for transaction status
   if confirmed -> mark reward claimed regardless of API error
}
```

2. Use Transaction-Coupled Confirmation
3. Add Post-Failure Reconciliation

TEAM

Fixed in trepa-onchain-svm PR #139 and trepa-api PR #289. The team went beyond the original recommendation: instead of only shifting DB updates to the indexer, an on-chain replay-protection PDA seeded with `(reward_id, streak_id, wallet)` was added so the chain itself rejects duplicate claims. The API now passes `reward_id` to `claimStreakReward` and lets the indexer drive the DB state from the emitted event with strict checks on the mapping.

[M-04] Non-qualifying streak participants are not reset until someone wins

Scope note: This is outside the originally narrowed API scope if only web3.service.ts is considered, but it affects backend reward eligibility and can create invalid claimable streak rewards.

`processStreakWinners` is responsible for maintaining each user's consecutive streak count. A user who meets the pool's precision threshold should have `current_streak_count` incremented, while a user who fails the threshold should have `current_streak_count` reset to `0`.

The function handles the "nobody qualifies" case correctly by resetting all participants. However, it mishandles the mixed case where at least one participant qualifies, but nobody reaches `streak_count_required`.

In that path, the function increments qualifying users, checks whether anyone has reached the required streak count, and returns early if there are no winners. This early return happens before the later reset block for non-qualifying participants.

The current ordering is:

```
increment qualifying users
check winners
return if no winners
reset non-qualifying users
```

Because the function returns before the reset step when `winningUsers.size === 0`, failed participants are only reset in rounds where someone else wins a streak reward.

The expected ordering should be:

```
increment qualifying users
reset non-qualifying users
check winners
return if no winners
```

This guarantees that every participant's streak state is updated for the current pool before the function decides whether to create streak rewards or return early.

Example

Assume:

- `streak_count_required = 2`
- `min_precision_score_required = 250`
- User A already has `current_streak_count = 1` after qualifying in pool 1

Round flow:

Pool 1 — User A qualifies, no other winners yet. - Expected `current_streak_count` for A: **1** - Actual: **1** (correct)

Pool 2 — User A fails the threshold; User B qualifies, but nobody reaches `streak_count_required`. - Expected `current_streak_count` for A: **0** (must reset because A did not qualify) - Actual: **1** (bug — A is not reset because the function returns early when no winners are produced)

Pool 3 — User A qualifies again; other users do not matter. - Expected `current_streak_count` for A: **1** (a fresh streak starting from Pool 3) - Actual: **2** (stale Pool 1 count + Pool 3 increment — A is now incorrectly treated as a streak winner)

In pool 3, User A is incorrectly treated as a streak winner even though their real sequence was:

```
qualify -> fail -> qualify
```

That is not a consecutive streak.

Impact

This can create invalid `streak_rewards` rows for users who did not satisfy the streak rules. These

reward rows are claimable through the normal streak reward flow, so the issue can move value out of the streak pot to ineligible users.

The backend distributes half of the available streak balance among detected streak winners:

```
const distributableAmount = new Decimal(availableAmount)
  .mul(0.5)
  .toNumber();

const amountPerWinner = new Decimal(distributableAmount)
  .div(winningUsers.size)
  .toNumber();
```

So the impact is not limited to incorrect UI state. The backend can allocate real claimable rewards to users who failed the consecutive-streak requirement and then reduce `available_balance` based on those invalid winners.

Root Cause

The reset logic for non-qualifying users is placed after the no-winner early return:

```
const winningUsers = new Set(winnersRows.map((r) => r.user_wallet_address));

if (winningUsers.size === 0) {
  return {
    qualifying_users: qualifying_users_with_count,
    non_qualifying_users: [],
    streak_winners: [],
  };
}
```

The reset block below is skipped in that branch:

```
const nonQualifyingParticipantWallets = [...participantWalletsSet].filter(
  (w) => !qualifyingWallets.includes(w),
);

if (nonQualifyingParticipantWallets.length > 0) {
  await tx
    .update(progressTable)
    .set({
      current_streak_count: 0,
      last_streak_event: StreakEvent.RESET,
    })
    .where(
      and(
        eq(progressTable.streak_id, streak.id),
        inArray(
          progressTable.user_wallet_address,
          nonQualifyingParticipantWallets,
        ),
      ),
    );
}
```

Location

- trepa-api/libs/shared/src/services/streak-processing.service.ts:211-220
- trepa-api/libs/shared/src/services/streak-processing.service.ts:261-292

RECOMMENDATION

Reset non-qualifying participants before checking for the no-winner early return.

A safer flow is:

```
increment qualifying users
reset non-qualifying users
check winners
create rewards or return no winners
```

The fix can be implemented by moving the existing non-qualifying reset block above the `winningUsers.size === 0` branch.

Also update the early return to return the actual reset users:

```
if (winningUsers.size === 0) {
  return {
    qualifying_users: qualifying_users_with_count,
    non_qualifying_users: non_qualifying_users_to_notify,
    streak_winners: [],
  };
}
```

As a defensive improvement, restrict the winner query to users who qualified in the current pool:

```
const winnersRows = await tx
  .select({
    user_wallet_address: progressTable.user_wallet_address,
  })
  .from(progressTable)
  .where(
    and(
      eq(progressTable.streak_id, streak.id),
      inArray(progressTable.user_wallet_address, qualifyingWallets),
      gte(progressTable.current_streak_count, streak_count_required),
    ),
  );
```

This prevents stale or unrelated `streak_progress` rows from being considered winners unless the user actually qualified in the current pool.

TEAM

Fixed in trepa-api PR #272. The reset step was moved before the early-return check so non-qualifying participants are reset every round (not only when someone wins), and the winner query was additionally restricted to wallets that qualified in the current pool.

[M-05] Pool can be permanently bricked when a winner's leaf exceeds `max_roi`

On-chain `claim_rewards` enforces a per-winner ROI cap. If a claim's `user_gains` exceed `stake * max_roi / 10000`, the instruction reverts with `MaxRoiExceeded`:

```
if amount >= ctx.accounts.prediction.stake {
   let user_gains = amount.checked_sub(ctx.accounts.prediction.stake)...;
   let gain_cap_amount = stake_u128
      .checked_mul(max_roi_u128)?
      .checked_div(10000)?;
   ...
   if user_gains > gain_cap_amount {
      return Err(CustomError::MaxRoiExceeded.into());
   }
}
```

However the resolver side has no corresponding guard:

1. `finalizePool.ts` only checks `sum(prizes) + protocolFee <= vault_balance`. It does not fetch per-winner stakes, nor does it validate each `winner.prize - stake <= stake * max_roi / 10000`.
2. `Web3Service.resolvePool` in `trepa-api/libs/shared/src/services/web3.service.ts` forwards the prizes array straight into `finalizePool`/`finalizeStreakPool` without touching `max_roi`.
3. The WASM distribution engine (`trepa-api/libs/wasm/rust/src/lib.rs`) caps per-winner gain with a **hardcoded** `max_roi_cap() = 100`, not the pool's actual `max_roi`. So any `POOL_MAX_ROI` env config that does not match the WASM constant silently produces leaves that will revert on-chain.

Because the merkle proof binds to the exact leaf amount, an over-cap winner cannot claim a smaller amount — every attempt reverts. `pool.claims_left` therefore never reaches 0, and:

```
require!(pool.claims_left == 0, CustomError::ClaimsPending);
```

in `close_pool_account` makes the pool un-closable. The vault is stuck and the over-cap winner is fully locked out.

Example:

1. Creator sets `max_roi = 100k` (10x) at pool creation
2. 1000 predictors stake 1 USDC each -> vault = 1000 USDC
3. The price moves a lot in an unexpected direction, resulting in 5 winners
4. Resolver's distribution logic ("split pool equally among winners") produces leaves of 200 USDC each
5. Per winner cap is `1 * 100k / 10k = 100 USDC` gain -> max leaf = 100 USDC
6. Every winner claim reverts with `MaxRoiExceeded`
7. `claims_left` stays at 5, `close_pool_account` reverts, vault's 1000 USDC is permanently stranded

The main problem here is that an attacker is not required and this bug can happen on its own.

RECOMMENDATION

Enforce the per-winner cap at tree construction time, and bind the WASM distribution to the pool's real `max_roi`.

TEAM

Fixed; the WASM reward calculator now consumes the pool's configured `max_roi` instead of a hard-coded `100x` cap, so leaves can no longer be constructed above the on-chain `MaxRoiExceeded` boundary that would previously have stranded the vault.

[M-06] Token-2022 ATA derivation mismatch can DoS `finalize_pool` on time-series pools

`finalize_pool` for time-series pools passes the streak token account in `remaining_accounts[1]` and validates it with `assert_valid_token_account`, which compares the passed account to a *derived* "canonical" ATA. That helper uses legacy SPL-style `get_associated_token_address(authority, mint)` for the expected address. For **Token-2022** mints, the **correct** associated token address depends on the **token program**; derivation without the token program yields a **different** pubkey than the real Token-2022 ATA.

Proof of Concept:

```
import * as anchor from '@coral-xyz/anchor';
import { AnchorError, Program } from '@coral-xyz/anchor';
import NodeWallet from '@coral-xyz/anchor/dist/cjs/nodewallet';
import {
  getAssociatedTokenAddress,
  TOKEN_2022_PROGRAM_ID,
  TOKEN_PROGRAM_ID,
} from '@solana/spl-token';
import { Keypair, PublicKey, sendAndConfirmTransaction } from '@solana/web3.js';
import { BN } from 'bn.js';
import { expect } from 'chai';

import {
  confirmTransactionWithTimeout,
  createPrediction,
  createStreakPool,
  finalizeStreakPool,
  getByteArray,
  getConfigPDA,
  getPoolPDAandIdArray,
  getStreakAccumulatorPDA,
  registerStreak,
  sendAndConfirmTx,
} from '../../utils';
import { scaleToInteger } from '../../utils/helpers/decimal';
import { Trepa } from '../../utils/idl/trepa';
import { GLOBAL_PLATFORM_FEE, SELF_CLAIM_WINDOW } from '../helpers/const';
import { createTokenAccount, Token, transferToken } from '../helpers/token';
import {
  generateRandomId,
  getCreator,
  getResolver,
  getTreasury,
} from '../helpers/utils';

describe('PoC: Token-2022 streak ATA derivation mismatch', () => {
```

```
anchor.setProvider(anchor.AnchorProvider.env());
const program = anchor.workspace.trepa as Program<Trepa>;
const provider = anchor.getProvider();
const connection = provider.connection;
const programWallet = (program.provider.wallet as NodeWallet).payer;

let treasury: Keypair;
let aliceCreator: Keypair;
let bobResolver: Keypair;
let usdt22: Token;
let fundingAta: PublicKey;

const streakId = generateRandomId();
const poolId = generateRandomId();

const minStake = 1;
const maxStake = 10;
const minOutcome = 1;
const maxOutcome = 100;
const precision = 2;
const step = 1;
const maxRoi = 5000;
const streakFeeBps = 500;

// Step 0 — setup: mint Token-2022 USDT, funding ATA, initialize global
// config if needed.
// `initialize` requires `admin` == the pubkey baked into the program at
// compile time (`ADMIN_PUBKEY`
// when running `anchor build`). If you see Unauthorized on init,
// rebuild with:
//   ADMIN_PUBKEY=$(solana-keygen pubkey ~/.config/solana/id.json) anchor
// build
before(async () => {
  treasury = await getTreasury();
  aliceCreator = await getCreator();
  bobResolver = await getResolver();

  const envAdmin = process.env.ADMIN_PUBKEY;
  if (envAdmin && envAdmin !== programWallet.publicKey.toBase58()) {
    throw new Error(
      `ADMIN_PUBKEY (${envAdmin}) must match the Anchor provider wallet (${
        programWallet.publicKey.toBase58()}).`,
    );
  }

  usdt22 = await Token.builder()
    .name('USDT')
    .tokenIndex(2)
    .decimals(9)
    .tokenProgram(TOKEN_2022_PROGRAM_ID)
    .build(connection, programWallet);

  const createdFundingAta = await createTokenAccount(
    connection,
    usdt22.mint,
    programWallet,
    usdt22.tokenProgram,
    usdt22.decimals,
  );
  if (!createdFundingAta) {
    throw new Error('Failed to create funding ATA for program wallet');
```

```
  }
  fundingAta = createdFundingAta;

  const configPDA = getConfigPDA(program);
  const existingConfig = await connection.getAccountInfo(configPDA);
  if (!existingConfig) {
    try {
      await program.methods
        .initialize(
          new BN(GLOBAL_PLATFORM_FEE),
          treasury.publicKey,
          aliceCreator.publicKey,
          bobResolver.publicKey,
          new BN(SELF_CLAIM_WINDOW),
        )
        .accounts({
          admin: programWallet.publicKey,
          program: program.programId,
        })
        .rpc();
    } catch (e: unknown) {
      const msg = e instanceof Error ? e.message : String(e);
      const unauthorized =
        (e instanceof AnchorError &&
          e.error.errorCode.code === 'Unauthorized') ||
        msg.includes('Error Code: Unauthorized') ||
        msg.includes('Error Number: 6007');
      if (unauthorized) {
        throw new Error(
          [
            'initialize() failed with Unauthorized: admin must match
               ADMIN_PUBKEY embedded at `anchor build`.',
            'Rebuild the program so it matches your Anchor wallet,
               then rerun tests:',
            '  ADMIN_PUBKEY=$(solana-keygen pubkey ~/.config/solana/id.json)
               anchor build',
          ].join('\n'),
        );
      }
      throw e;
    }
  }
});

it('Alice creates Token-2022 streak pool; Bob cannot finalize due to ATA
   mismatch', async () => {
  // Flow: setup (before) -> register -> pool -> fund -> predict -> wait ->
  // derive ATAs -> build finalize -> submit (expect InvalidAtaData).
  // Step 1 — register: creator registers the streak (accumulator PDA is tied
  // to this id).
  const registerTx = await registerStreak(
    program,
    aliceCreator.publicKey,
    streakId,
    streakFeeBps,
  );
  await sendAndConfirmTx(connection, aliceCreator, registerTx, 'confirmed');

  const now = await connection.getBlockTime(await connection.getSlot());
  if (now === null) {
    throw new Error('Failed to fetch block time');
```

```
}

// Step 2 — pool: open a streak pool using Token-2022 (correct
// token_program in pool state).
const createPoolTx = await createStreakPool(
  program,
  aliceCreator.publicKey,
  usdt22.mint,
  poolId,
  new BN(now + 2),
  new BN(now + 10),
  new BN(scaleToInteger(minStake, usdt22.decimals)),
  new BN(scaleToInteger(maxStake, usdt22.decimals)),
  new BN(maxRoi),
  new BN(scaleToInteger(minOutcome, precision)),
  new BN(scaleToInteger(maxOutcome, precision)),
  precision,
  streakId,
  scaleToInteger(step, precision),
  usdt22.tokenProgram,
);
await sendAndConfirmTx(connection, aliceCreator, createPoolTx, 'confirmed');
await new Promise((resolve) => setTimeout(resolve, 2500));

// Step 3 — fund: send USDT to the resolver so they can stake.
await transferToken(
  connection,
  usdt22.tokenProgram,
  usdt22.mint,
  fundingAta,
  programWallet,
  bobResolver,
  scaleToInteger(3, usdt22.decimals),
  usdt22.decimals,
);

// Step 4 — predict: resolver places a prediction (stake uses the
// Token-2022 vault path).
const predictTx = await createPrediction(
  program,
  connection,
  bobResolver.publicKey,
  poolId,
  new BN(scaleToInteger(3, usdt22.decimals)),
  new BN(scaleToInteger(10, precision)),
  usdt22.mint,
  usdt22.tokenProgram,
  bobResolver.publicKey,
);
const vpredictTx = anchor.web3.VersionedTransaction.deserialize(
  Buffer.from(predictTx, 'base64'),
);
vpredictTx.sign([bobResolver]);
const predictSig = await connection.sendTransaction(vpredictTx);
const { blockhash, lastValidBlockHeight } =
  await connection.getLatestBlockhash();
await confirmTransactionWithTimeout(
  connection,
  predictSig,
  blockhash,
  lastValidBlockHeight,
```

```
);

// Step 5 — wait: until the pool window ends so finalize is allowed.
await new Promise((resolve) => setTimeout(resolve, 9000));

const winners = [
 {
  address: bobResolver.publicKey,
  prize: scaleToInteger(2, usdt22.decimals),
 },
];

// Step 6 — derive ATAs: SPL vs Token-2022 give different addresses for the
// same (mint, owner).
const streakAccumulatorPDA = getStreakAccumulatorPDA(
 program,
 getByteArray(streakId),
);
const token2022Ata = await getAssociatedTokenAddress(
 usdt22.mint,
 streakAccumulatorPDA,
 true,
 TOKEN_2022_PROGRAM_ID,
);
const legacySplAta = await getAssociatedTokenAddress(
 usdt22.mint,
 streakAccumulatorPDA,
 true,
 TOKEN_PROGRAM_ID,
);
expect(token2022Ata.equals(legacySplAta)).to.equal(false);

// Predictions fund the pool vault — not the streak accumulator ATA. The
// latter may only appear in
// finalize pre-instructions (`createAssociatedTokenAccount` in
// finalizeStreakPool). Sanity-check stake.
const { poolPDA } = getPoolPDAandIdArray(program, poolId);
const poolVaultAta = await getAssociatedTokenAddress(
 usdt22.mint,
 poolPDA,
 true,
 TOKEN_2022_PROGRAM_ID,
);
const poolVaultInfo = await connection.getAccountInfo(poolVaultAta);
expect(
 poolVaultInfo,
 'pool vault ATA should exist after predict (stake is here,
    not at streak accumulator)',
).to.not.equal(null);

const [infoStreak2022, infoStreakLegacy] = await Promise.all([
 connection.getAccountInfo(token2022Ata),
 connection.getAccountInfo(legacySplAta),
]);

// Step 7 — build finalize: client passes the Token-2022 streak ATA;
// instruction accounts are checked below.
const finalizeTx = await finalizeStreakPool(
 program,
 bobResolver.publicKey,
 poolId,
```

```
  winners,
  new BN(scaleToInteger(1, usdt22.decimals)),
  usdt22.mint,
  usdt22.tokenProgram,
  usdt22.decimals,
  streakId,
);

const finalizeInstruction = finalizeTx.instructions.find((ix) =>
  ix.programId.equals(program.programId),
);
expect(finalizeInstruction).to.not.equal(undefined);
const passedAccountKeys =
  finalizeInstruction?.keys.map((meta) => meta.pubkey.toBase58()) ?? [];
expect(passedAccountKeys).to.include(token2022Ata.toBase58());
expect(passedAccountKeys).to.not.include(legacySplAta.toBase58());

// Step 8 — submit finalize and expect InvalidAtaData (on-chain ATA
// validation vs program expectation).
let logs: string[] = [];
try {
  await sendAndConfirmTransaction(connection, finalizeTx, [bobResolver]);
  expect.fail('finalize should revert');
} catch (err) {
  const message = (err as Error)?.message ?? '';
  logs =
    typeof (err as any)?.getLogs === 'function'
      ? (((await (err as any).getLogs()) as string[]) ?? [])
      : Array.isArray((err as any)?.logs)
        ? ((err as any).logs as string[])
        : [];
  if (logs.length === 0) {
    throw new Error(`finalize failed but no logs (message: ${message})`);
  }
}

const anchorErr = AnchorError.parse(logs);
expect(anchorErr, 'expected Anchor error in transaction logs').to.not.equal(
  null,
);

// This is the core PoC assertion:
// finalize receives a Token-2022 ATA (not legacy SPL ATA),
// but the program still rejects it
// as InvalidAtaData due to ATA derivation mismatch in helper validation.
expect(anchorErr!.error.errorCode.code).to.equal('InvalidAtaData');
expect(anchorErr!.error.errorCode.number).to.equal(6054);
expect(anchorErr!.error.errorMessage).to.equal(
  'Invalid associated token account data',
);

console.log(
  '[PoC] Finalize reverted with InvalidAtaData',
  `(code ${anchorErr!.error.errorCode.number}):`,
  anchorErr!.error.errorMessage,
  '| streak ATA exists (Token-2022 / legacy):',
  infoStreak2022 !== null,
  '/',
  infoStreakLegacy !== null,
  '| token-2022 streak ATA:',
  token2022Ata.toBase58(),
```

```
    );
  });
});
```

PoC Output

```
  PoC: Token-2022 streak ATA derivation mismatch
Creating new mint for USDT on localnet...
Creating token mint with decimals: 9 and owner:
   EAJhp1R734MjFrjEPfP3LuqVQs6ncf5wzoUuHPehCnX5
Transfered tokens to E6QJqPJe32zh67Ubpp1cRneqiaLse36XLYgwSBP2QnPn:
   5gGY6BoB3ERPtLVS2TR9BTNqUQnanZd7cPpRUWz5Ptowj6jityxxZrLHZBTjFQAxBpYxdyj8V61c
   q3N8Cn349TB
[PoC] Finalize reverted with InvalidAtaData (code 6054): Invalid associated
   token account data | streak ATA exists (Token-2022 / legacy): false / false
   | token-2022 streak ATA: 6UBjhkprsHDJc4JtudTfPrdY4xPj5qgVahZxqFt1KhFk
   ✔ Alice creates Token-2022 streak pool; Bob cannot finalize due to ATA
      mismatch (13854ms)
```

Scenario: Alice (creator) opens a time-series pool with a Token-2022 stake mint; Bob (resolver) calls finalize with the **correct** Token-2022 streak ATA; validation still fails, so **finalization (and downstream claims flow) is blocked** — operational DoS for those pools.

RECOMMENDATION

In `assert_valid_token_account`, derive the expected ATA with **token-program-aware** associated-token derivation (e.g. `get_associated_token_address_with_program_id` / equivalent that takes **`token_program`** / Token-2022 program id), not legacy SPL-only derivation.

```
- let expected_ata = get_associated_token_address(authority.key,
   token_mint.key);
+ let expected_ata = get_associated_token_address_with_program_id(
+    authority.key,
+    token_mint.key,
+    token_program.key,
+ );
```

TEAM

Fixed in trepa-onchain-svm PR #115. The canonical ATA derivation in `assert_valid_token_account` was switched from `get_associated_token_address` to `get_associated_token_address_with_program_id`, so Token-2022 mints derive the correct address. Trepa does not currently use Token-2022 mints in production, but the fix lands defensively to avoid the trap if support is added later.

Low Severity

[L-01] Access tokens remain valid for 41 days due to millisecond expiry passed as JWT seconds

The API intends access sessions to last one hour, but the JWT `exp` is set roughly `1000x` longer than the access cookie lifetime.

The shared constant is named and defined in milliseconds:

```
// trepa-api/libs/shared/src/constants/auth.constants.ts:1-3
export const ACCESS_TOKEN_EXPIRY_MS = 60 * 60 * 1000;

export const REFRESH_TOKEN_EXPIRY_MS = 7 * 24 * 60 * 60 * 1000;
```

That value is correct for the Express cookie `maxAge`, which expects milliseconds:

```
// trepa-api/apps/api/src/auth/auth.config.ts:62-67
get accessTokenCookieOptions(): CookieOptions {
 return {
  ...this.cookieOptions,
  maxAge: ACCESS_TOKEN_EXPIRY_MS,
 };
}
```

However, the same millisecond value is also passed to JWT signing as `expiresIn`:

```
// trepa-api/apps/api/src/auth/auth.config.ts:10-18
get jwtSignOptions(): JwtSignOptions {
 return {
  secret: this.configService.getOrThrow<string>('JWT_SECRET'),
  expiresIn: ACCESS_TOKEN_EXPIRY_MS,
  issuer: 'trepa-api',
  audience: 'trepa-app',
  algorithm: 'HS256',
 };
}
```

`@nestjs/jwt` uses `jsonwebtoken`, and numeric `expiresIn` values are interpreted as seconds, not milliseconds. Since Trepa passes `60 * 60 * 1000`, the signed JWT receives:

```
expiresIn = 3,600,000 seconds
3,600,000 / 86,400 = 41.6667 days
```

I confirmed this against the pinned local `jsonwebtoken@9.0.3` dependency:

```
node -e "const jwt=require('./node_modules/.pnpm/jsonwebtoken@9.0.3/node_modules
  /jsonwebtoken'); const t=jwt.sign({id:'u'}, 'secret', {expiresIn:
  60*60*1000, issuer:'trepa-api', audience:'trepa-app'}); const d=jwt.decode(
  t); console.log('jsonwebtoken_exp_minus_iat_seconds=' + (
  d.exp-d.iat)); console.log('jsonwebtoken_days=' + ((d.exp-d.iat)/86400));"
```

Output:

```
jsonwebtoken_exp_minus_iat_seconds=3600000
jsonwebtoken_days=41.666666666666664
```

Login and API-key session creation both mint this access token and place it in the `trepa-token` cookie:

```
// trepa-api/apps/api/src/auth/auth.controller.ts:56-72
async login(@Req() req: AuthenticatedRequest, @Res() res: Response) {
  const accessToken = this.authService.generateJwt(req.user);
  const refreshToken = await this.authService.generateRefreshToken(
    req.user.id,
  );

  res
    .cookie(
      'trepa-token',
      accessToken,
      this.authConfig.accessTokenCookieOptions,
    )
    .cookie(
      'trepa-refresh',
      refreshToken,
      this.authConfig.refreshTokenCookieOptions,
    );

  return res.send();
}
```

```
// trepa-api/apps/api/src/auth/auth.controller.ts:90-106
async loginApiKey(@Req() req: AuthenticatedRequest, @Res() res: Response) {
  const accessToken = this.authService.generateJwt(req.user);
  const refreshToken = await this.authService.generateRefreshToken(
    req.user.id,
  );

  res
    .cookie(
      'trepa-token',
      accessToken,
      this.authConfig.accessTokenCookieOptions,
    )
    .cookie(
      'trepa-refresh',
      refreshToken,
      this.authConfig.refreshTokenCookieOptions,
    );

  return res.send();
}
```

The browser cookie expires after one hour, so the normal browser happy path mostly behaves as intended. The problem is that the token value itself remains cryptographically valid for about 41.7 days. A copied token can be replayed by setting it as the `trepa-token` cookie until its JWT `exp` is reached.

The JWT strategy only extracts from the `trepa-token` cookie and enforces the JWT `exp`:

```
// trepa-api/apps/api/src/auth/strategies/jwt.strategy.ts:11-22
super({
  jwtFromRequest: ExtractJwt.fromExtractors([
    (req: Request) => req?.cookies?.['trepa-token'] ?? null,
  ]),
  secretOrKey: configService.getOrThrow<string>('JWT_SECRET')!,
  ignoreExpiration: false,
  passReqToCallback: false,
  issuer: 'trepa-api',
  audience: 'trepa-app',
  algorithms: ['HS256'],
});
```

The JWT strategy validates the token and returns the claims from the token payload as the authenticated user. It does not reload the user from the database on each request:

```
// trepa-api/apps/api/src/auth/strategies/jwt.strategy.ts:31-44
const {
  iat: _iat,
  exp: _exp,
  aud: _aud,
  iss: _iss,
  ...redactedUser
} = payload;

return redactedUser;
```

So any copied access token continues to represent that same user state until the token expires. This includes stale role claims, but the core issue is simply the access-token lifetime being much longer than the cookie/session lifetime.

Normal case: browser users lose the `trepa-token` cookie after one hour, so the frontend prompts them to refresh/re-login as expected.

Worst case: any copied `trepa-token` value remains usable as a cookie for about 41.7 days instead of one hour, extending authenticated API access for that user far beyond the intended access-token window.

This is Low severity because it does not directly bypass signatures, steal funds, or bypass JWT verification. It is still a real auth-window bug: access tokens remain valid far longer than the product's cookie lifetime and naming indicate.

RECOMMENDATION

Use separate units for JWT expiry seconds and cookie expiry milliseconds, and make the JWT value impossible to confuse with the cookie value.

Replace `trepa-api/libs/shared/src/constants/auth.constants.ts` with:

```
export const ACCESS_TOKEN_EXPIRY_SECONDS = 60 * 60;
export const ACCESS_TOKEN_EXPIRY_MS = ACCESS_TOKEN_EXPIRY_SECONDS * 1000;

export const REFRESH_TOKEN_EXPIRY_MS = 7 * 24 * 60 * 60 * 1000;
```

Then replace the import and access-token JWT signing option in `trepa-api/apps/api/src/auth/auth.config.ts`:

```
import {
  ACCESS_TOKEN_EXPIRY_MS,
  ACCESS_TOKEN_EXPIRY_SECONDS,
  REFRESH_TOKEN_EXPIRY_MS,
} from '@/libs/shared';
```

```
get jwtSignOptions(): JwtSignOptions {
  return {
    secret: this.configService.getOrThrow<string>('JWT_SECRET'),
    expiresIn: ACCESS_TOKEN_EXPIRY_SECONDS,
    issuer: 'trepa-api',
    audience: 'trepa-app',
    algorithm: 'HS256',
  };
}
```

Keep the cookie option using milliseconds:

```
get accessTokenCookieOptions(): CookieOptions {
  return {
    ...this.cookieOptions,
    maxAge: ACCESS_TOKEN_EXPIRY_MS,
  };
}
```

Also add a regression test that signs an access token and asserts:

```
decoded.exp - decoded.iat == 3600
```

That test catches the exact unit regression and makes sure future refactors do not accidentally feed millisecond constants back into JWT `expiresIn`.

TEAM

Fixed in trepa-api PR #282. The JWT `expiresIn` value is now converted from milliseconds to seconds before being passed to the signer, restoring the intended one-hour access-token lifetime.

[L-02] Pyth volatility calculation accepts partial candle data and can distort streak precision scores

`PythService.computeStdLogReturns()` is used to compute the BTC volatility input (`std_log_returns`) that the WASM reward calculator uses for `precision_score`. That precision score is later used by `StreakProcessingService` to decide whether a participant qualifies for streak progress and whether a claimable `streak_rewards` row is created.

The issue is that the backend intends to compute 7-day BTC 1-minute standard deviation of log returns, but it does not validate that Pyth returned a complete or recent 7-day 1-minute candle set. The only explicit coverage check is `closes.length < 2`; in practice, any tiny close set that produces

finite variance, such as three valid closes, is accepted as a 7-day sample.

This is not just a stale-cache/design-choice issue. The stronger bug is that an under-covered or partial Pyth OHLC response is treated as a valid 7-day volatility sample.

Buggy code in `trepa-api/libs/shared/src/services/pyth.service.ts`:

```
const startTs = endTs - this.SEVEN_DAYS_SEC;

const dayRanges: { from: number; to: number }[] = [];

let current = startTs;

while (current < endTs) {
  const dayEnd = Math.min(current + this.ONE_DAY_SEC, endTs);
  dayRanges.push({ from: current, to: dayEnd });
  current = dayEnd + 1;
}
```

The intended range is seven days. At 1-minute resolution, that should produce roughly:

```
7 * 24 * 60 = 10,080 close prices
```

However, the response handling ignores timestamps and only collects close prices:

```
const closes: number[] = [];
for (const { data } of results) {
  if (data.s !== 'ok' || !data.c?.length) {
    this.logger.warn(
      `Pyth OHLC chunk returned no data (s=${data.s}), skipping`,
    );
    continue;
  }
  for (const c of data.c) {
    const close = Number(c);
    if (!Number.isFinite(close) || close <= 0) continue;
    closes.push(close);
  }
}

if (closes.length < 2) {
  throw new Error('Pyth 7d OHLC: insufficient data for std(log returns)');
}
```

The code does not verify:

- `data.t.length === data.c.length`
- the returned timestamps cover the requested 7-day range
- each daily chunk has close to 1,440 one-minute candles
- the latest returned candle is near the requested `endTs`
- the final total is close to 10,080 closes

It then computes volatility from whatever close list survived:

```
const logReturns = closes
  .slice(0, -1)
  .map((_, i) => Math.log(closes[i + 1] / closes[i]));
const n = logReturns.length;
const mean = logReturns.reduce((a, b) => a + b, 0) / n;
const variance =
  logReturns.reduce((acc, x) => acc + (x - mean) ** 2, 0) / (n - 1);
const std = Math.sqrt(variance);
```

That value is passed into reward processing during pool resolution:

```
const stdLogReturnValue =
  await this.stdLogReturnsService.getLatestBtcPythValueOrPopulate();

const result = await this.rewardsMechanismService.process(
  predictions.map((p) => ({
    address: p.predictor_account,
    stake: p.stake,
    value: p.prediction,
  })),
  pool.min_outcome,
  pool.max_outcome,
  outcomeValue,
  stdLogReturnValue,
);
```

In `trepa-api/libs/wasm/rust/src/lib.rs`, `std_log_returns` directly changes `precision_score`:

```
let scoring_sensitivity = Decimal::from_str("2").unwrap().ln(
    ) / std_log_returns_dec;

for (address, log_err) in log_return_errors {
    let exponent = Decimal::ZERO - scoring_sensitivity * log_err;
    let exp_val = exponent.checked_exp().unwrap_or(Decimal::ZERO);
    let ps_dec = (precision_score_scale() * exp_val).max(precision_score_floor(
       ));
    let ps_f = ps_dec
       .to_f64()
       .ok_or(CustomError::DecimalToF64ConversionFailed)?;
    ps.insert(address.clone(), ps_f);
}
```

Higher `std_log_returns` makes scoring more forgiving. Lower `std_log_returns` makes scoring harsher.

The precision score is then used for streak qualification:

```
if (participant.precision_score >= minPrecisionRequiredNum) {
  qualifying_users.push({
    user_wallet_address: participant.user_wallet_address,
    precision_score: participant.precision_score,
  });
} else {
  non_qualifying_users.push({
    user_wallet_address: participant.user_wallet_address,
```

```
    precision_score: participant.precision_score,
  });
}
```

Qualifying users have their streak count incremented:

```
await tx
  .update(progressTable)
  .set({
    current_streak_count: sql`${progressTable.current_streak_count} + 1`,
    last_streak_event: StreakEvent.QUALIFIED,
  })
  .where(
    and(
      eq(progressTable.streak_id, streak.id),
      inArray(progressTable.user_wallet_address, qualifyingWallets),
    ),
  );
```

If the updated streak count reaches `streak_count_required`, the backend creates claimable streak reward rows:

```
const streakRewardsToCreate = walletAddressesArray.map((walletAddress) => ({
  streak_id: streak.id,
  user_wallet_address: walletAddress,
  amount: amountPerWinnerRounded,
  is_claimed: false,
}));

await tx
  .insert(permissionedSchema.streakRewardsTable)
  .values(streakRewardsToCreate);
```

So a partial or stale volatility value can change user-visible streak state and streak reward accounting, even though it does not change normal pool winner selection.

Illustrative Example

Assume:

```
prediction = 105
outcome = 100
min_precision_score_required = 250
current_streak_count = 2
streak_count_required = 3
available streak balance = 1000
```

The same 5% prediction error produces different precision scores depending only on `std_log_returns`:

```
std_log_returns = 0.02 -> precision_score = 184.346... -> does not qualify
std_log_returns = 0.05 -> precision_score = 508.456... -> qualifies
```

With the higher volatility value, the user reaches streak count 3 and receives a claimable streak

reward for 50% of the available streak balance.

Proof of Concept

Run:

```
npx --yes tsx pyth_stdlog_precision_repro.ts threshold
npx --yes tsx pyth_stdlog_precision_repro.ts coverage
```

Full PoC code:

```
/**
 * Standalone demonstrator for the Pyth/std-log-return precision-score risk.
 *
 * No NestJS, DB, WASM, or chain dependencies. This mirrors the relevant math
 * and streak-threshold behavior from:
 *  trepa-api/libs/wasm/rust/src/lib.rs
 *  trepa-api/libs/shared/src/services/pyth.service.ts
 *  trepa-api/libs/shared/src/services/streak-processing.service.ts
 */

const PRECISION_SCORE_SCALE = 1_000;
const PRECISION_SCORE_FLOOR = 100;
const EXPECTED_SEVEN_DAY_ONE_MINUTE_CLOSES = 7 * 24 * 60;

function precisionScore(
  prediction: number,
  outcome: number,
  stdLogReturns: number,
): number {
  if (prediction <= 0 || outcome <= 0 || stdLogReturns <= 0) {
    throw new Error("prediction, outcome, and stdLogReturns must be positive");
  }

  const logError = Math.abs(Math.log(prediction) - Math.log(outcome));
  const scoringSensitivity = Math.log(2) / stdLogReturns;
  const score = PRECISION_SCORE_SCALE * Math.exp(-scoringSensitivity *
    logError);
  return Math.max(PRECISION_SCORE_FLOOR, score);
}

function simulatedStreakResult(args: {
  precisionScore: number;
  minPrecisionRequired: number;
  currentStreakCount: number;
  streakCountRequired: number;
  availableBalance: number;
}) {
  const qualifies = args.precisionScore >= args.minPrecisionRequired;
  const updatedCount = qualifies ? args.currentStreakCount + 1 : 0;
  const winsStreak = qualifies && updatedCount >= args.streakCountRequired;
  const createdStreakReward = winsStreak ? args.availableBalance * 0.5 : 0;

  return {
    qualifies,
    updatedCount,
    winsStreak,
    createdStreakReward,
  };
}
```

```
function computeStdLogReturnsFromCloses(closes: number[]): number {
  const validCloses = closes
    .map(Number)
    .filter((close) => Number.isFinite(close) && close > 0);

  if (validCloses.length < 2) {
    throw new Error("Pyth 7d OHLC: insufficient data for std(log returns)");
  }

  const logReturns = validCloses
    .slice(0, -1)
    .map((_, i) => Math.log(validCloses[i + 1] / validCloses[i]));
  const n = logReturns.length;
  const mean = logReturns.reduce((a, b) => a + b, 0) / n;
  const variance =
    logReturns.reduce((acc, x) => acc + (x - mean) ** 2, 0) / (n - 1);
  const std = Math.sqrt(variance);

  if (!Number.isFinite(std) || std <= 0) {
    throw new Error("Pyth 7d: invalid std computed");
  }

  return std;
}

function thresholdProof() {
  const prediction = 105;
  const outcome = 100;
  const minPrecisionRequired = 250;
  const currentStreakCount = 2;
  const streakCountRequired = 3;
  const availableBalance = 1_000;

  const accurateStdLogReturns = 0.02;
  const staleOrBadStdLogReturns = 0.05;

  const accurateScore = precisionScore(
    prediction,
    outcome,
    accurateStdLogReturns,
  );
  const staleScore = precisionScore(
    prediction,
    outcome,
    staleOrBadStdLogReturns,
  );

  return {
    proof: "same prediction/outcome, different std_log_returns flips streak
       qualification",
    inputs: {
      prediction,
      outcome,
      logReturnError: Math.abs(Math.log(prediction) - Math.log(outcome)),
      minPrecisionRequired,
      currentStreakCount,
      streakCountRequired,
      availableBalance,
    },
    withAccurateStdLogReturns: {
```

```
    stdLogReturns: accurateStdLogReturns,
    precisionScore: accurateScore,
    streak: simulatedStreakResult({
     precisionScore: accurateScore,
     minPrecisionRequired,
     currentStreakCount,
     streakCountRequired,
     availableBalance,
    }),
   },
   withStaleOrBadStdLogReturns: {
    stdLogReturns: staleOrBadStdLogReturns,
    precisionScore: staleScore,
    streak: simulatedStreakResult({
     precisionScore: staleScore,
     minPrecisionRequired,
     currentStreakCount,
     streakCountRequired,
     availableBalance,
    }),
   },
   verdict:
    "A stale/incorrectly high std_log_returns value turns the same 5%
       prediction error from non-qualifying into qualifying and creates a
       streak reward.",
  };
}

function coverageProof() {
  const tinyCoverageCloses = [100, 101, 103];
  const computedStd = computeStdLogReturnsFromCloses(tinyCoverageCloses);

  return {
   proof: "Pyth std computation accepts far less than a 7-day 1-minute window",
   expectedSevenDayOneMinuteCloses: EXPECTED_SEVEN_DAY_ONE_MINUTE_CLOSES,
   providedCloses: tinyCoverageCloses.length,
   coverageRatio:
    tinyCoverageCloses.length / EXPECTED_SEVEN_DAY_ONE_MINUTE_CLOSES,
   computedStdLogReturns: computedStd,
   verdict:
    "The computation can return a std_log_returns value from only a few
       closes because it validates only close count/value,
       not timestamp coverage or latest candle freshness.",
  };
}

function main() {
  const mode = process.argv[2] ?? "threshold";
  if (mode === "threshold") {
   console.log(JSON.stringify(thresholdProof(), null, 2));
   return;
  }
  if (mode === "coverage") {
   console.log(JSON.stringify(coverageProof(), null, 2));
   return;
  }
  if (mode === "all") {
   console.log(
    JSON.stringify(
     {
      threshold: thresholdProof(),
```

```
      coverage: coverageProof(),
    },
    null,
    2,
   ),
  );
  return;
 }

 throw new Error(`Unknown mode: ${mode}`);
}

main();
```

Observed output for `threshold` mode:

```
{
 "proof": "same prediction/outcome, different std_log_returns flips streak
   qualification",
 "inputs": {
  "prediction": 105,
  "outcome": 100,
  "logReturnError": 0.04879016416943127,
  "minPrecisionRequired": 250,
  "currentStreakCount": 2,
  "streakCountRequired": 3,
  "availableBalance": 1000
 },
 "withAccurateStdLogReturns": {
  "stdLogReturns": 0.02,
  "precisionScore": 184.34648220180247,
  "streak": {
   "qualifies": false,
   "updatedCount": 0,
   "winsStreak": false,
   "createdStreakReward": 0
  }
 },
 "withStaleOrBadStdLogReturns": {
  "stdLogReturns": 0.05,
  "precisionScore": 508.4566617977571,
  "streak": {
   "qualifies": true,
   "updatedCount": 3,
   "winsStreak": true,
   "createdStreakReward": 500
  }
 },
 "verdict": "A stale/incorrectly high std_log_returns value turns the same 5%
   prediction error from non-qualifying into qualifying and creates a streak
   reward."
}
```

Observed output for `coverage` mode:

```
{
 "proof": "Pyth std computation accepts far less than a 7-day 1-minute window",
 "expectedSevenDayOneMinuteCloses": 10080,
```

```
  "providedCloses": 3,
  "coverageRatio": 0.00029761904761904765,
  "computedStdLogReturns": 0.006829336666098421,
  "verdict": "The computation can return a std_log_returns value from only a
     few closes because it validates only close count/value,
     not timestamp coverage or latest candle freshness."
}
```

Impact, Likelihood, and Severity

This can create incorrect streak qualification and streak reward state.

Possible consequences:

- users who should fail `min_precision_score_required` can incorrectly qualify
- users who should qualify can incorrectly fail and be reset
- users can incorrectly reach `streak_count_required`
- the backend can create claimable `streak_rewards` for users who should not have earned them
- legitimate streak participants can miss rewards they should have received

This does not change the base pool winner set or base pool reward weights. The impact is specifically on the precision-score and streak-reward layer.

- Impact: Low. This does not affect base pool winner selection or base pool reward weights. The impact is limited to streak qualification, streak progress, and possible streak reward-row creation.
- Likelihood: Low. Normal users do not directly control Pyth responses. The issue depends on incomplete upstream data, stale cached values, transient API failures, or infrastructure/proxy conditions. The backend should still fail closed on incomplete volatility data because stale or under-covered data can distort streak state, but the bug is a data-integrity / reward-accounting hardening issue rather than a direct exploit primitive.

RECOMMENDATION

Do not fix this only by changing the cache TTL. The cache window is a product choice; the concrete bug is that the value can be computed from incomplete candle data. The resolver should fail closed unless the Pyth response proves it covers the intended 7-day 1-minute window.

In `trepa-api/libs/shared/src/services/pyth.service.ts`, replace the entire `computeStdLogReturns()` method with the implementation below.

This replacement:

- rejects non-`ok` Pyth chunks instead of silently skipping them
- verifies `t[]` and `c[]` are present and have matching lengths
- sorts and deduplicates candles by timestamp
- requires at least 95% coverage of the expected 7-day 1-minute window
- verifies the first and last candles are close to the requested window boundaries
- computes log returns only from validated, timestamp-ordered closes

Replacement code:

```
async computeStdLogReturns(
  symbol: StdLogReturnSymbol,
```

```
  resolution: StdLogReturnResolution,
): Promise<number> {
  if (resolution !== StdLogReturnResolution.ONE_MINUTE) {
    throw new Error(`Unsupported std log return resolution: ${resolution}`);
  }

  const RESOLUTION_SECONDS = 60;
  const MIN_EXPECTED_COVERAGE_RATIO = 0.95;
  const MAX_BOUNDARY_DRIFT_SECONDS = RESOLUTION_SECONDS * 2;

  const now = new Date();
  const endTs =
    Date.UTC(
      now.getUTCFullYear(),
      now.getUTCMonth(),
      now.getUTCDate(),
      0,
      0,
      0,
      0,
    ) / 1000;
  const startTs = endTs - this.SEVEN_DAYS_SEC;

  const dayRanges: { from: number; to: number }[] = [];
  let current = startTs;

  while (current < endTs) {
    const dayEnd = Math.min(current + this.ONE_DAY_SEC, endTs);
    dayRanges.push({ from: current, to: dayEnd });
    current = dayEnd;
  }

  const results = await Promise.all(
    dayRanges.map(({ from, to }) =>
      firstValueFrom(
        this.httpService.get<PythOhlcResponse>(this.API_URL, {
          params: {
            symbol,
            resolution: String(resolution),
            from,
            to,
          },
          timeout: 30_000,
        }),
      ),
    ),
  );

  const candles: Array<{ timestamp: number; close: number }> = [];

  for (const { data } of results) {
    if (!data || typeof data !== 'object') {
      throw new Error('Pyth OHLC chunk returned malformed response');
    }

    if (data.s !== 'ok') {
      throw new Error(`Pyth OHLC chunk returned non-ok status: ${data.s}`);
    }

    if (!Array.isArray(data.t) || !Array.isArray(data.c)) {
      throw new Error('Pyth OHLC chunk is missing timestamp or close arrays');
```

```
  }

  if (data.t.length !== data.c.length) {
    throw new Error(
      `Pyth OHLC timestamp/close length mismatch: ` +
        `t=${data.t.length} c=${data.c.length}`,
    );
  }

  for (let i = 0; i < data.c.length; i++) {
    const timestamp = Number(data.t[i]);
    const close = Number(data.c[i]);

    if (!Number.isFinite(timestamp)) continue;
    if (!Number.isFinite(close) || close <= 0) continue;
    if (timestamp < startTs || timestamp > endTs) continue;

    candles.push({ timestamp, close });
  }
}

candles.sort((a, b) => a.timestamp - b.timestamp);

const dedupedCandles: Array<{ timestamp: number; close: number }> = [];
for (const candle of candles) {
  const previous = dedupedCandles[dedupedCandles.length - 1];
  if (previous && previous.timestamp === candle.timestamp) {
    dedupedCandles[dedupedCandles.length - 1] = candle;
    continue;
  }
  dedupedCandles.push(candle);
}

const expectedCount = Math.floor((endTs - startTs) / RESOLUTION_SECONDS);
const minCount = Math.floor(expectedCount * MIN_EXPECTED_COVERAGE_RATIO);

if (dedupedCandles.length < minCount) {
  throw new Error(
    `Pyth OHLC coverage too low: got ${dedupedCandles.length}, ` +
      `expected at least ${minCount}`,
  );
}

const first = dedupedCandles[0];
const last = dedupedCandles[dedupedCandles.length - 1];

if (!first || first.timestamp > startTs + MAX_BOUNDARY_DRIFT_SECONDS) {
  throw new Error('Pyth OHLC first candle is too far from requested start');
}

if (!last || last.timestamp < endTs - MAX_BOUNDARY_DRIFT_SECONDS) {
  throw new Error('Pyth OHLC latest candle is too stale for requested end');
}

const closes = dedupedCandles.map((c) => c.close);
const logReturns = closes
  .slice(0, -1)
  .map((_, i) => Math.log(closes[i + 1] / closes[i]));

if (logReturns.length < 2) {
  throw new Error('Pyth 7d OHLC: insufficient data for std(log returns)');
```

```
  }

  const n = logReturns.length;
  const mean = logReturns.reduce((a, b) => a + b, 0) / n;
  const variance =
    logReturns.reduce((acc, x) => acc + (x - mean) ** 2, 0) / (n - 1);
  const std = Math.sqrt(variance);

  if (!Number.isFinite(std) || std <= 0) {
    throw new Error('Pyth 7d: invalid std computed');
  }

  this.logger.log(
    `BTC 1m std log returns (Pyth 7d): ${std.toExponential(4)} ` +
      `(samples=${dedupedCandles.length}, logReturns=${logReturns.length})`,
  );

  return std;
}
```

After deploying this change, invalidate or recompute existing rows in `std_log_returns`, because rows created before the patch may have been computed from under-covered data. The current `StdLogReturnsService` already fails closed during pool resolution when it cannot produce a value, because `getLatestBtcPythValueOrPopulate()` returns `null` on computation failure and `PoolsResolutionService` rejects missing/stale std log returns.

For stronger long-term safety, add coverage metadata to the cached DB row (`window_start`, `window_end`, and `sample_count`) and validate that metadata when reading from cache. That is an operational hardening layer; the minimum security fix is the `pyth.service.ts` replacement above.

TEAM

Fixed in trepa-api PR #285. The volatility resolver now requires Pyth's response to cover ≥95% of the expected 7-day 1-minute window (sample count ≈ 10080) and validates the boundary timestamps before computing log returns, so under-covered or transient responses are rejected instead of silently distorting the precision score.

[L-03] Pool vault donations can block the shipped finalization helper before resolver submission

The pool vault is an associated token account owned by the pool PDA, but it is still a normal SPL token account that anyone can transfer the pool mint into. The on-chain program accepts residual vault tokens: `close_pool_account` later sweeps any remaining vault balance to the treasury in `trepa-onchain-svm/programs/trepa/src/lib.rs:965`.

However, the shipped finalization helpers treat the live vault balance as an exact settlement invariant. In `trepa-onchain-svm/utils/instructions/finalizePool.ts:97`, the helper reads the pool token account balance. It then compares that balance against `sum(winner prizes) + protocolFee`, and at `trepa-onchain-svm/utils/instructions/finalizePool.ts:119` throws if the vault contains more than the expected settlement by more than `ALLOWED_PRECISION_DEVIATION`. The streak finalization helper repeats the same check at `trepa-onchain-svm/utils/instructions/finalizePool.ts:258-L282`.

This path is live through `trepa-api/libs/shared/src/services/web3.service.ts:323`, where `resolvePool` calls `finalizePool` / `finalizeStreakPool` before the resolver transaction is submitted.

As a result, an external user can transfer a small amount of the stake mint directly into a pool vault before resolution and cause the normal backend finalization path to fail before it reaches the on-chain `finalize_pool` instruction.

Impact

Pool resolution can be griefed through the shipped backend/helper path. For a 6-decimal USDC pool, transferring slightly more than `10000` raw units, about one cent, into the pool vault is enough to make the helper reject otherwise-valid settlement data. Funds are not stolen, and operators can recover by manually bypassing or fixing the helper, but automatic resolution can be blocked until intervention.

Likelihood

Low to Medium. The attack is public and cheap once the pool vault exists, but it is a liveness grief rather than a direct value-extraction path, and recovery is operationally possible.

RECOMMENDATION

In `trepa-onchain-svm/utils/instructions/finalizePool.ts`, replace the duplicated `if (totalClaimAmount < stake)` block in both functions: `finalizePool` and `finalizeStreakPool`.

```
if (totalClaimAmount < stake) {
  const difference = stake - totalClaimAmount;

  if (difference > ALLOWED_PRECISION_DEVIATION) {
    throw new Error(
      `Pool total stake: ${stake}, total claim amount: ${totalClaimAmount}.
        Difference: ${difference} is greater than allowed precision
        deviation: ${ALLOWED_PRECISION_DEVIATION}. Error in calculations.`,
    );
  }

  const originalProtocolFee = protocolFeeNumber;
  protocolFeeNumber += difference;

  console.log(
    `Pool total stake ${stake} was greater than the total claim amount ${
      totalClaimAmount}. Protocol fee adjusted from ${originalProtocolFee} to
      ${protocolFeeNumber}.`,
  );
}
```

with:

```
if (totalClaimAmount < stake) {
  const difference = stake - totalClaimAmount;

  console.log(
    `Pool vault has ${difference} extra raw token units; leaving them as
      residual for close_pool_account treasury sweep.`,
  );
}
```

This preserves the existing underfunded-vault rejection while preventing excess public donations from blocking finalization.

TEAM

Fixed in trepa-onchain-svm PR #138 (bundled with the L-06 cap-protocol-fee work). The settlement check no longer compares against the live vault balance; instead, a new `total_stake` field is stored on the pool and incremented inside `predict`/`update_stake`, and the helper compares against that. External donations to the vault remain as residual and are swept later by `close_pool_account`, so a 1-cent donation can no longer brick finalization.

[L-04] Pool finalization can commit unreachable claim counts and permanently deadlock cleanup

`finalize_pool` accepts resolver-supplied terminal settlement data and stores the supplied `claims` scalar directly into `pool.claims_left` after only checking that `claims > 0`. From that point onward, cleanup assumes the resolver-supplied claim count exactly matches the set of reachable on-chain claims that can be consumed through `claim_rewards`.

The missing invariant is that `claims_left` must equal the number of uniquely reachable, still-unconsumed on-chain claims. That invariant is never enforced. Successful claiming is enforced one `PredictionAccount` at a time: each successful `claim_rewards` call verifies one Merkle proof, marks `prediction.is_claimed = true`, decrements `claims_left` once, and auto-closes that prediction account. Final pool cleanup still requires `claims_left == 0`. If accepted finalization data overstates the number of actually claimable payouts, every reachable winner can finish and the pool can still remain permanently finalized but unclosable.

The shipped path already narrows this because the API and PDA model both enforce one prediction per `(pool, predictor)` pair. But the resolution backend still forwards `result.winners` into on-chain finalization without asserting that the committed winner set is unique and that its claim count matches the set of DB rewards and reachable prediction accounts. The cleanup jobs then use DB reward and `is_closed` state as their precondition, so the backend can believe cleanup is complete while the chain still refuses it because `claims_left` never reached zero.

Impact: Low The affected pool can still be stuck in a terminal state where cleanup, rent reclamation, and residual sweep logic never complete.

Likelihood: Low This is not a public no-privilege exploit, and the shipped path needs internally inconsistent resolver-submitted finalization state before the deadlock can materialize.

Location:

- `trepa-onchain-svm/programs/trepa/src/lib.rs:391-413`
- `trepa-onchain-svm/programs/trepa/src/lib.rs:782-856`
- `trepa-onchain-svm/programs/trepa/src/lib.rs:928-945`
- `trepa-api/libs/shared/src/services/pools-resolution.service.ts:336-347`
- `trepa-api/libs/shared/src/services/close-prediction-accounts.service.ts:76-86`
- `trepa-api/libs/shared/src/services/close-pool.service.ts:51-64`

RECOMMENDATION

- Immediate mitigation: in `programs/trepa/src/lib.rs:409`, reject any finalization where `claims > pool.open_predictions_left` before writing `pool.claims_left = claims`.

TEAM

Fixed in trepa-onchain-svm PR #131 and trepa-api PR #287. The on-chain `finalize_pool` now rejects any submission where `claims > open_predictions_left`, and the resolver path additionally deduplicates winners, rejects zero/negative prizes, and verifies that each winner's `PredictionAccount` exists and is unclaimed before committing the Merkle root (with `getMultipleAccountsInfo` chunked into batches of 100 to handle large pools).

[L-05] Multiple privileged operational roles are concentrated behind one backend signing boundary

The on-chain program intentionally separates privileged actions across different authorities. `FinalizePool` and `ClaimStreakRewards` require the configured resolver to sign, while post-window reward relays and pool close depend on the configured creator path. However, the in-scope backend service `web3.service.ts` holds all four operational signers at once: `creatorKeypair`, `resolverKeypair`, `feeSponsorKeypair`, and `streakTopupKeypair`.

That same service then uses those signers across multiple privileged flows. The resolver path is used for pool finalization and streak-reward relaying. The creator path is used for creator-co-signed presigned prediction, stake-update, and reward-claim transactions, as well as pending-claim and cleanup flows. Separate fee-sponsor and streak-topup authorities are also exercised from the same service boundary for withdraw sponsorship and streak funding. The shared helper in `transaction.ts` then adds the backend signer before broadcast.

This issue is narrower than saying the program's authorization model is broken. The on-chain checks are working as designed, and admin freeze/role-rotation can contain the incident once detected. The real problem is that one in-scope backend signing service concentrates several distinct privileged domains that the protocol otherwise separates on chain.

Impact: Low Most of the exposed roles are operational or treasury-adjacent, and can be contained by freeze or rotation once detected. The strongest lasting risk is resolver misuse before containment, because a malicious finalization or streak-claim authorization can land on chain before the signer is rotated.

Likelihood: Low This is not permissionless, and exploitation requires compromise of the backend signing boundary plus acting before operators freeze the program or rotate the affected role. Location:

- `trepa-api/libs/shared/src/services/web3.service.ts:94-114`
- `trepa-api/libs/shared/src/services/web3.service.ts:295-355`
- `trepa§-api/libs/shared/src/services/web3.service.ts:505-533`
- `trepa-api/libs/shared/src/services/web3.service.ts:660-770`
- `trepa-api/libs/shared/src/services/web3.service.ts:782-824`
- `trepa-api/libs/shared/src/services/web3.service.ts:998-1080`
- `trepa-onchain-svm/utils/helpers/transaction.ts:262-281`
- `trepa-onchain-svm/programs/trepa/src/context.rs:301-345`
- `trepa-onchain-svm/programs/trepa/src/context.rs:543-586`
- `trepa-onchain-svm/programs/trepa/src/lib.rs:720-738`
- `trepa-onchain-svm/programs/trepa/src/context.rs:460-509`

RECOMMENDATION

Prioritize the resolver path first. This only meaningfully helps if it creates a real isolation boundary, such as a separate worker, deployment, or signer backend with its own secret scope, rather than just another class inside the same Nest process.

- Primary fix: move `resolvePool`, `constructClaimStreakRewardTransaction`, and `sendPresignedClaimStreakRewardTransaction` into a resolver-only worker or service that is injected with `resolverKeypair` and no other signer.
- Secondary blast-radius reduction: move `constructWithdrawTransaction` and `sendPresignedWithdrawTransaction` into a fee-sponsor-only service, move `topUpStreakPotAmount` into a streak-topup-only service, and keep creator-side relay and cleanup helpers such as `constructPendingClaimTransaction`, `constructClosePredictionAccountTransaction`, `signAndSendCreatorTransaction`, and the presigned creator relay methods in a creator-only service.
- Any code-level wrapper or module split only helps if it is backed by real runtime separation and separate secret scope; splitting methods across classes inside the same process is mostly organizational, not a meaningful security boundary.

[L-06] `platform_fee` is dead config, while `protocol_fee` is unbounded at finalization

The code stores and validates `platform_fee` during `initialize`, but this value is never enforced anywhere in runtime fee flow:

- `initialize` validates and stores `config.platform_fee`.
- `finalize_pool` accepts a caller-supplied `protocol_fee` parameter.
- `finalize_pool` does **not** check `protocol_fee` against `config.platform_fee`, pool balance ratio, or any basis-point bound.
- Fee transfers are executed directly from pool vault to treasury (and optional streak split), based solely on the passed `protocol_fee`.

So the effective protocol fee is not governed by config at all. It is decided by resolver input. This makes `platform_fee` misleading/dead state.

Impact analysis

- **Economic policy bypass:** protocol operators/users may assume `platform_fee` is the global cap, but on-chain it is unenforced.
- **Over-fee risk from resolver path:** a malicious/compromised resolver (or buggy integration) can set arbitrarily high `protocol_fee` up to available vault balance.

RECOMMENDATION

Either remove the `protocol_fee` instruction argument and derive the fee entirely from the config, or keep it but strictly cap it by the config and document it as “requested_fee, capped by policy.”

TEAM

Fixed across trepa-onchain-svm PRs #138, #140, #142 and trepa-api PRs #288, #293. Rather than removing the `protocol_fee` argument, the team kept it (needed for refund-style finalizations where `protocol_fee = 0`) and added an on-chain cap of `pool.total_stake * pool.platform_fee / 10000`, returning a new `ProtocolFeeTooHigh` error. Two follow-up cycles were needed: first to exclude max-ROI `leftover` from the on-chain settlement comparison so capped-winner residual stays in

the vault for `close_pool_account` to sweep, then to source the WASM fee rate from the pool's stored `platform_fee` instead of a hardcoded 20% so non-default fee rates don't trip the new cap.

[L-07] Unsafe account sizing via `std::mem::size_of` in init constraints can break future upgrades

The program allocates account space using Rust in-memory layout (`std::mem::size_of::<T>()`) instead of Anchor/Borsh serialized layout (e.g., `T::INIT_SPACE`) in multiple `init` constraints (`ConfigAccount`, `PoolAccount`, `PredictionAccount`, `StreakAccumulatorAccount`).
Today, these structs are mostly fixed-size, so impact is currently limited. However, this is a high-risk upgrade footgun: the codebase already includes upgrade-oriented fields (`_reserved`). If future upgrades introduce variable-length fields (`String`, `Vec`, etc.), `size_of` can under-allocate accounts and cause initialization/write failures (DoS of affected flows).

RECOMMENDATION

Replace all `space = 8 + std::mem::size_of::<T>()` with Anchor canonical sizing: - Derive `InitSpace` on account structs. - Use `space = 8 + T::INIT_SPACE` in every `init`.

TEAM

Fixed in trepa-onchain-svm PR #130. All `init` constraints were switched from `8 + std::mem::size_of::<T>()` to Anchor's canonical `8 + T::INIT_SPACE` (with `#[derive(InitSpace)]` on the affected accounts), eliminating the future-upgrade footgun if variable-length fields are ever added to those structs.