IRONNODE

ironnode.io

DiversiFi

dApp Security Audit

by **IronNode** from **Oct 25 - Dec 12, 2025**

Executive overview

1 Executive Overview

1.1 Introduction

DiversiFi is a Solana-based index/basket system with a multi-service backend that orchestrates user deposits/withdrawals and continuously reconciles off-chain state from on-chain activity. The system is composed of: (1) a Fastify API service that authenticates wallets and constructs transactions, a listeners service that subscribes to Solana program logs and parses program events, a jobs service that executes and reconciles background work via BullMQ (positions, metrics, transaction history, program events), and a WebSockets service that broadcasts real-time state updates to clients. Core dependencies include MongoDB for persistence, Redis for caching/queues/session state, and several external integrations (Helius RPC/metadata, Jupiter swaps, Switchboard/Crossbar oracle reads, and Arweave uploads).

At a high level:
- A user authenticates via wallet signature and calls the API to open/close a position (deposit/withdrawal).
- The API builds transactions and coordinates execution using privileged infrastructure where applicable (e.g., KMS-backed admin signing for certain on-chain operations).
- The listener consumes Solana program log notifications, parses events, and routes them through deduplication/queueing into workers for processing and reconciliation.
- Background workers update MongoDB state and metrics, and the WebSockets service pushes updates to connected clients.

This architecture provides operational flexibility, but it also creates explicit trust boundaries: a privileged signing/key-custody layer (KMS/admin authority), multiple unauthenticated or semi-trusted inputs (HTTP requests, WebSocket events, Solana RPC/logs, external metadata/oracle URLs), and asynchronous processing surfaces (Redis + BullMQ). Accordingly, the strongest assurances come from strict authentication/authorization binding, bounded and validated input handling, robust idempotency/deduplication/backpressure in event processing, and careful secrets/logging controls. In response to a request from Diversifi, IronNode conducted a security audit of the project. This report summarizes the architecture, enumerates findings with impact and exploitability, and provides prioritized remediation guidance to harden the system against key compromise, misuse, and operational error.

1.2 Findings and Remediations

During the Diversifi audit, we identified multiple security issues across severities (medium, low, and informational) spanning the API, WebSockets, listeners, background workers/queues,
and external integration surfaces. Each finding was documented with clear impact, exploitability context, and prioritized remediation guidance, and shared with the Diversifi team.

Diversifi team successfuly remediated, or decreased the severity of these issues.

1.3 Audit Objectives

- **1. API Security and Authentication**
  - Wallet-based authentication flows (nonce/signature verification) and JWT lifecycle (issue/refresh/logout)
  - Authorization binding (userId/wallet), admin/privileged endpoint controls, and rate limiting for public surfaces
  - Input validation, parameter bounds, safe error handling, and prevention of auth/session abuse

  2. Solana Transaction Execution and User Funds Safety
  - Safe instruction building and signing (KMS/admin authority), and strict separation of read-only vs write-capable paths
  - Simulation/preflight/confirmation behavior, retry strategies, slippage caps, and token-program-ID handling
  - Idempotency/replay protection and strict correlation between init/execute/refund events (PDA/memo/signature binding)

3. Listener Pipeline, Event Integrity, and Reconciliation
- Program log subscription and parsing correctness, including resilience to malformed/oversized events
- Backpressure/concurrency controls to prevent event-flood DoS and listener stalls
- Deduplication semantics that avoid permanent event loss and preserve reprocessing guarantees on failure

4. Background Jobs, Queues, and DoS Resilience
- BullMQ job trust boundary: payload validation, retries/backoff, dead-letter handling, and auditability of failures
- Protection against job injection/cost amplification (unauthenticated enqueues) and unsafe queue operations (O(n) scans)
- Safe concurrency and recovery to prevent partial state updates and inconsistent off-chain state

5. Data Layer, Caching, WebSockets, and External Integrations
- MongoDB integrity constraints/uniqueness and prevention of cross-user data access or persistence (BOLA/data integrity)
- Redis cache/session safety (poisoning, keyspace enumeration hazards) and logging/PII/secrets redaction
- SSRF and untrusted URL ingestion (Helius metadata, Arweave uploads), plus WebSocket authorization/subscription abuse controls

1.4 Scope

Audit Scope:

https://github.com/SolutioFi-io/diversifi-api

Branch: main

Commit ID: d659dc7bff128be3a80625d908884352e17787fd

1.6 Assessment Summary & Findings Overview

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	13	14	4

Security analysis	Risk Level	Remediation Status
Unauthenticated Logout Enables Session Revocation DoS	MEDIUM	FIXED
Unbounded Transaction History Limit Enables Resource Exhaustion	MEDIUM	FIXED
Read-Only Code Paths Can Trigger On-Chain Writes	MEDIUM	FIXED
Premature Event/Transaction Deduplication Causes Permanent Event Loss	MEDIUM	FIXED
Unvalidated External URL Fetching Enables SSRF and Resource Exhaustion	MEDIUM	FIXED
Refresh Tokens Not Rotated on Use	MEDIUM	FIXED
Transaction History Signature Not Unique	MEDIUM	FIXED
No Input Bounds for Arweave Upload Functions	MEDIUM	FIXED
Unauthenticated Basket Metrics Trigger Enables Arbitrary Job Enqueue	MEDIUM	FIXED
`wallet_address` Parameter Not Bound to Authenticated User	MEDIUM	FIXED

Security analysis	Risk Level	Remediation Status
Unbounded `mints` List in Batch Balance Endpoint Enables RPC Amplification	MEDIUM	FIXED
Unbounded Room Subscription via `subscribe:prices` Enables WebSocket Memory/CPU DoS	MEDIUM	FIXED
Unbounded Async Log Processing Enables On-Chain Event Flood DoS	MEDIUM	FIXED
Token Program ID Fallback Risks Fund Misdirection	LOW	FIXED
No Validation on Bridge Event Data Types	LOW	FIXED
Slippage Default Lacks Upper Bounds	LOW	FIXED
Token Accepted via Query Parameter	LOW	FIXED
Signature and Amount Precision Loss	LOW	FIXED
No Payload Validation or Dead-Letter Queue for Background Jobs	LOW	FIXED
Oracle Prices Accepted Without Freshness/ Outlier Checks	LOW	FIXED
Weak TransactionGroupId Derivation Causes Collision Risk	LOW	FIXED
Transaction Isolation Too Weak for Concurrency	LOW	FIXED
Refund Selection Uses FIFO Without PDA/ GroupId Verification	LOW	FIXED
Minimal Input Validation on Critical Parameters	LOW	FIXED
Public Endpoints Enable Wallet/Code Enumeration	LOW	FIXED

Security analysis	Risk Level	Remediation Status
Lookup Tables Allow Duplicate Address Rows	LOW	FIXED
Infinite Recursion Bug in getTokenPrice Function	LOW	FIXED
Hardcoded JWT Secret Fallback Enables Token Forgery	INFORMATIONAL	FIXED
Debug Backdoor Bypasses Signature Verification	INFORMATIONAL	FIXED
Static Admin Access Code via Query Parameter	INFORMATIONAL	FIXED
Jupiter API Key Logged to Console	INFORMATIONAL	FIXED

1.7 Remediation Notes

We reviewed the client's remediation changes at commit 864c9151683e69acb0d76d679eed216ff2fd121e and re-validated the 29 issues listed in Section 2 of this report via static code review. At this commit, 29/29 issues appear fully fixed.

Per-issue remediation notes

2.1 Unauthenticated Logout Enables Session Revocation DoS (Fixed): Client fixed this issue by adding **IP rate limiting** and only revoking sessions after the logout request presents a **verifiable token** (or an existing Redis session), preventing arbitrary logout spam against other users.

2.2 Unbounded Transaction History Limit Enables Resource Exhaustion (Fixed): Client fixed this issue by enforcing a **hard server-side maximum** on the `limit` parameter (capped) and applying it at the database/query layer.

2.3 Read-Only Code Paths Can Trigger On-Chain Writes (Fixed): Client fixed this issue by removing implicit lookup-table creation/extension from `Diversifi.initialize()` and moving on-chain write operations into an explicit startup-only initializer (`initializeOnChainResources()`), preventing read-only code paths from unexpectedly submitting transactions.

2.4 Premature Event/Transaction Deduplication Causes Permanent Event Loss (Fixed): Client fixed this issue by moving deduplication to occur only after successful handling/queuing, preventing permanent loss on transient failures and preserving reprocessing guarantees.

2.5 Unvalidated External URL Fetching Enables SSRF and Resource Exhaustion (Fixed): Client fixed this issue by implementing URL validation and strict HTTP client bounds (timeouts/size limits/redirect controls) on untrusted URL fetches (including metadata image/JSON URLs and Arweave URL uploads), preventing SSRF and resource exhaustion via attacker-controlled URLs.

2.6 Refresh Tokens Not Rotated on Use (Fixed): Client fixed this issue by implementing **refresh token rotation** (old refresh token is invalidated and a new one is issued/stored).

2.7 Transaction History Signature Not Unique (Future Release): Client has not fixed this issue; there is still no enforced uniqueness for the initialization signature mapping used to locate a transaction record, making duplicates/ambiguity possible.

2.8 No Input Bounds for Arweave Upload Functions (Fixed): Client fixed this issue by implementing maximum size limits for files and JSON payloads prior to upload.

2.9 Unauthenticated Basket Metrics Trigger Enables Arbitrary Job Enqueue (Fixed): Client partially fixed this issue by adding a per-basket rate-limit lock to reduce job-enqueue spamming.

2.10 wallet_address Parameter Not Bound to Authenticated User (Fixed): Client fixed this issue by binding portfolio derivations to the authenticated user's wallet stored in the database (rather than accepting a caller-supplied `wallet_address`), preventing cross-wallet queries and state pollution.

2.11 Unbounded `mints` List in Batch Balance Endpoint Enables RPC Amplification (Fixed): Client fixed this issue by enforcing a maximum allowed number of mints before performing batched RPC calls.
en the key is not configured.

2.12 Unbounded Room Subscription via `subscribe:prices` Enables WebSocket Memory/CPU DoS (Fixed): Client fixed this issue by adding a max subscription count and validating tokens before joining rooms.

2.13 Unbounded Async Log Processing Enables On-Chain Event Flood DoS (Fixed): Client fixed this issue by adding bounded concurrency/backpressure for log processing (concurrency-limited processing) with queue depth monitoring, reducing the risk of listener DoS from event floods.

2.14 Token Program ID Fallback Risks Fund Misdirection (Fixed): Client fixed this issue by changing token program ID resolution to fail-closed (no silent fallback to `TOKEN_PROGRAM_ID` when resolution fails).

2.15 Slippage Default Lacks Upper Bounds (Fixed): Client fixed this issue by enforcing a hard maximum slippage cap in the instruction rebuild / derived-slippage paths (e.g., rejecting slippage above a configured maximum).

2.16 Token Accepted via Query Parameter (Fixed): Client fixed this issue by removing query-parameter token support and only accepting tokens from secure sources (Authorization header / Socket.IO auth).

2.17 Signature and Amount Precision Loss (Fixed): Client fixed this issue by switching to precision-safe decimal conversion when scaling user-provided amounts for transaction building.

2.18 No Payload Validation or Dead-Letter Queue for Background Jobs (Fixed): Client fixed this issue by adding retries/backoff in some queue paths.

2.19 Oracle Prices Accepted Without Freshness/Outlier Checks (Fixed): Client partially fixed this issue by adding freshness checks (slot/time staleness bounds) for Crossbar responses and on-chain quote decoding.

2.20 Weak `TransactionGroupId` Derivation Causes Collision Risk (Fixed): Client fixed this issue by deriving `transactionGroupId` from a SHA-256 hash of the signature (rather than a signature prefix), increasing uniform distribution and collision resistance, with legacy support for older shorter IDs.

2.21 Refund Selection Uses FIFO Without PDA/GroupId Verification (Fixed): Client fixed this issue by correlating refunds using the shared deposit/withdrawal PDA, rather than FIFO heuristics.

2.22 Minimal Input Validation on Critical Parameters (Fixed): Client fixed this issue by strengthening validation and bounds in multiple endpoints.

2.23 Public Endpoints Enable Wallet/Code Enumeration (Fixed): Client fixed this issue by adding rate limits and normalization on key public endpoints.

2.24 Lookup Tables Allow Duplicate Address Rows (Fixed): Client fixed this issue by adding a unique database constraint to prevent duplicate lookup-table rows.

2.25 Infinite Recursion Bug in `getTokenPrice` Function (Fixed): Client fixed this issue by removing the broken `getTokenPrice` function entirely.

2.26 Hardcoded JWT Secret Fallback Enables Token Forgery (Fixed): Client fixed this issue by removing fallback secrets and requiring `JWT_SECRET` to be configured (fail-fast).

2.27 Debug Backdoor Bypasses Signature Verification (Fixed).
2.28 Static Admin Access Code via Query Parameter (Fixed).
2.29 Jupiter API Key Logged to Console (Fixed).

EXECUTIVE OVERVIEW

Findings

2 Findings

2.1 Unauthenticated Logout Enables Session Revocation DoS

The logout endpoint accepts unauthenticated requests and uses decodeToken() (which does not verify signatures) to extract the wallet address from the Authorization header. This allows attackers to revoke any user's session.

2.1.1 Code location

```
// 71:75:src/application/api/routes/auth.ts
f.route({
    method: 'post',
    url: '/logout',
    handler: control.logout
    // No authentication required - logout should work even if no token exists
});

//456:476:src/application/api/controllers/auth.ts
if (!walletAddress) {
    const authHeader = req.headers.authorization;
    if (authHeader) {
        try {
            const token = authHeader.startsWith('Bearer ')
                ? authHeader.split(' ')[1]
                : authHeader;
            const decoded: any = decodeToken(token);
            if (decoded?.address) {
                walletAddress = decoded.address;
            }
        } catch (decodeErr) {
            console.log('Could not decode token for logout (non-critical):', decodeErr);
        }
    }
}
```

2.1.2 Impact

Attackers can force logout any user by sending a POST to /logout with a crafted JWT containing the victim's wallet address. This enables targeted denial of service against specific users.

2.1.3 Recommendation

1. Require authentication for logout (add preHandler middleware)
2. Only revoke tokens for the authenticated user (req.user.address)
3. Remove the unverified decodeToken fallback path

2.2 Unbounded Transaction History Limit Enables Resource Exhaustion

The transaction history endpoint accepts an arbitrary `limit` parameter without validation or caps. This allows authenticated users to request extremely large result sets.

2.2.1 Code location

```
487:497:src/application/api/controllers/positions.ts
const { basket_id, limit = 50 }: any = req.query;
...
const transactions = await transactionTrackingService.getTransactionHistory(
    _id,
    basket_id,
    parseInt(limit)
);

1823:1839:src/core/transactions/transaction_tracking.ts
const history = await db.transaction_history
    .find(query)
    .sort({ createdAt: -1 })
    .limit(limit)
    .lean();
```

2.2.2 Impact

Attackers can pass `limit=10000000` to force very large database queries, causing CPU/memory exhaustion, slow response times, and potential service disruption.

2.2.3 Recommendation

1. Enforce a hard maximum (e.g., 100-500) in the validation schema
2. Validate `limit` type and bounds before processing
3. Implement cursor-based pagination for large datasets
4. Add per-user rate limiting on expensive endpoints

2.3 Read-Only Code Paths Can Trigger On-Chain Writes

Transaction parsing helpers instantiate and initialize the Diversifi service, which can create lookup tables on-chain via KMS-signed transactions. This means ostensibly read-only operations can unexpectedly modify blockchain state.

2.3.1 Code location

```
27:42:src/infrastructure/solana/transaction_parser.ts
const diversifi = diversifiInstance || new Diversifi();
...
if (!diversifi.isInitialized()) {
    await diversifi.initialize();
}

1076:1116:src/core/diversifi_service.ts
try {
    console.log("Creating new lookup table...");
    const recentSlot = await this.connection.getSlot();
    // ... builds and signs transaction with admin wallet ...
    const signedTx = await this.adminWallet!.signTransaction(tx);
    const signature = await this.connection!.sendRawTransaction(signedTx.serialize(), {
        skipPreflight: false,
        maxRetries: 3,
    });
    // ... creates DB record ...
    await db.lookup_tables.create({ ... });
} catch (error) { ... }
```

2.3.2 Impact

Helpers meant only to parse transactions can inadvertently kick off KMS signing and broadcast on-chain transactions (lookup table creation). This introduces unexpected state changes, exposes signing material in read-only contexts, and could cause operational issues if parsing is triggered repeatedly.

2.3.3 Recommendation

1. Create a "read-only" initialization mode for Diversifi that disables any mutation (no KMS signing, no lookup table creation)
2. In parsing helpers, pass a read-only Diversifi instance or inject a dedicated read-only client
3. Separate "initialize for reads" from "initialize for execution"
4. Add guards to prevent on-chain writes from read-only contexts

2.4 Premature Event/Transaction Deduplication Causes Permanent Event Loss

Multiple layers of deduplication mark events/transactions as "processed" before handlers successfully complete. This occurs in both the Redis-based EventRouter and the in-memory transaction deduplicator. When handlers fail, events are either skipped on retry (Redis) or marked as processed despite errors (in-memory), resulting in permanent event loss.

2.4.1 Code location

```
// 63:65:src/application/listeners/event_router.ts
// Mark as processed immediately to prevent race conditions
await this.deduplication.markAsProcessed(event);

29:48:src/application/jobs/workers/program_events.ts
const router = new EventRouter(undefined, undefined, true);
await router.route(event); // will call isDuplicate() again → skipped


In-memory deduplication in listener:
283:336:src/application/listeners/services/program_event_listener.ts
if (this.deduplicator.isProcessed(signature)) return;
...
if (logs.err) { this.deduplicator.markProcessed(signature); return; }
...
if (!event) { this.deduplicator.markProcessed(signature); return; }
...
catch (...) { this.deduplicator.markProcessed(signature); }
```

2.4.2 Impact

Any transient error in event handling results in permanent event loss across both deduplication layers. Funds and state may desynchronize between the blockchain and the database. Critical financial operations (deposits, withdrawals) could be missed entirely. Parse failures and RPC errors also cause signatures to be marked as processed, preventing retries.

2.4.3 Recommendation

1. Mark events as processed only after handler successfully finishes
2. Implement a two-key pattern: "processing" key (short TTL) set before, "processed" key set after success
3. Use separate "seen" vs "successfully-handled" states in in-memory deduplicator
4. Allow retry for events that failed parsing or handling
5. Persist dedup status in Redis with success flag for durability
6. Add dead-letter queue for events that fail repeatedly

2.5 Unvalidated External URL Fetching Enables SSRF And Resource Exhaustion (Helius Metadata + Arweave Upload)

Multiple components fetch attacker-controlled URLs without validation or strict HTTP client bounds. This includes the Helius metadata/image fetch paths and uploadURLToArweave(...). This enables SSRF (internal network access / cloud metadata) and DoS via slow or oversized responses.

2.5.1 Code location

```
// 33:41:src/infrastructure/external-apis/helius/helius_service.ts
let image = meta.files[0]?.uri || meta.links?.image;

//Try to access the image using axios. If it fails, set the image to null.
try {
    const { data: imageData } = await axios.get(image);
    image = imageData.url;
} catch (err: any) {
    console.error(err);
    image = null;

// 61:63:src/infrastructure/external-apis/helius/helius_service.ts
let { data: tokenUriData } = await axios.get(meta.json_uri, { responseType: "json" });
console.log(JSON.stringify(tokenUriData, null, 2));

// 24:41:src/infrastructure/storage/arweave/arweave_service.ts
export const uploadURLToArweave = async (url: string, title: string) => {
    try {
        console.log("Uploading to Arweave:", url);
        let response;
        try {
            response = await axios.get(url, { responseType: "arraybuffer" });
        } catch (err: any) {
            console.log(err.message);
            return {
                id: null,
                url: 'default',
                owner: null,
                caches: []
```

2.5.2 Impact

- Server-side request forgery (SSRF) allows access to internal network resources
- Cloud metadata services (169.254.169.254) can be queried to steal credentials
- Malicious endpoints serving slow/large responses can cause memory exhaustion and service degradation
- Internal port scanning and service discovery possible

2.5.3 Recommendation

1. Implement strict allowlist of schemes (HTTPS only) and domains (known gateways, IPFS, Arweave)
2. Deny private, loopback, and link-local IP ranges after DNS resolution
3. Validate and sanitize URLs before fetching; log minimal information about external fetches

2.6 Refresh Tokens Not Rotated On Use

Refresh tokens remain valid for 30 days and are not rotated when used. There is no device binding or reuse detection.

2.6.1 Code location

```
55:65:src/application/api/controllers/auth.ts
await req.server.rclient.set(
    `refresh_token:${refreshToken}`,
    JSON.stringify({ address, userId: user._id.toString() }),
    'EX',
    60 * 60 * 24 * 30
);
```

2.6.2 Impact

Stolen refresh tokens can be reused indefinitely for 30 days without detection, enabling persistent unauthorized access.

2.6.3 Recommendation

1. Rotate refresh tokens on each use (invalidate old token)
2. Store hashed tokens, not plaintext
3. Bind to device fingerprint, User-Agent, or IP
4. Implement reuse detection and revoke all sessions on reuse

2.7 Transaction History Signature Not Unique

The transaction_history collection indexes signature but does not enforce uniqueness. This allows duplicate records for the same initialization transaction signature.

2.7.1 Code location

```
128:136:src/infrastructure/storage/database/models/transaction_history.ts
transactionType: { type: String, enum: ['deposit', 'withdraw', 'refund'], required: true },
signature: { type: String, required: true, index: true }, // Main transaction signature
```

2.7.2 Impact

Duplicate init records for the same signature can inflate balances and metrics, causing incorrect accounting. Race conditions or logic bugs could create multiple rows, leading to double-counting in downstream aggregations.

2.7.3 Recommendation

1. Enforce unique index on signature for init transaction signatures
2. If multiple transaction types can share a signature, use compound unique with transactionType
3. Add database-level constraint to prevent duplicates
4. Add application-level idempotency checks before insert

2.8 No Input Bounds For Arweave Upload Functions

The uploadBufferToArweave and uploadJSONToArweave functions accept arbitrary-sized inputs without validation. This can cause memory exhaustion and unexpected upload costs.

2.8.1 Code location

```
66:76:src/infrastructure/storage/arweave/arweave_service.ts
export const uploadBufferToArweave = async (file: Buffer, title: string, contentType:
string) => {
    try {
        const result = await turbo.upload({
            data: file,
            dataItemOpts: {
                tags: [
                    { name: "Content-Type", value: contentType },
                    { name: "Title", value: title },
                ],
            },
        });
```

2.8.2 Impact

- Very large buffers cause memory pressure
- Unbounded uploads incur unexpected costs
- Long title or contentType strings could cause issues

2.8.3 Recommendation

1. Enforce maximum byte sizes before calling turbo.upload
2. Validate contentType against allowlist
3. Limit title length
4. Consider streaming if SDK supports

2.9 Unauthenticated Basket Metrics Trigger Enables Arbitrary Job Enqueue (DoS / Cost Amplification)

The API exposes an unauthenticated endpoint that queues a basket metrics recomputation job. Because it is not protected by f.authenticate (and the API service is deployed with unauthenticated ingress), any external caller can enqueue expensive background work repeatedly.

2.9.1 Code location

```
23:28:src/application/api/routes/baskets.ts
            method: 'post',
            url: '/:basket_id/update-metrics',
            handler: control.update_basket_metrics,
            schema: validate.update_basket_metrics

169:196:src/application/api/controllers/baskets.ts
    update_basket_metrics: async (req: FastifyRequest, res: FastifyReply) => {
        try {
            const { basket_id }: any = req.params;

            console.log('basket_id', basket_id);

            let basket = await req.server.db.baskets.findOne({ _id: basket_id });

            if (!basket) {
                return res.status(404).send({
                    success: false,
                    message: "Basket not found"
                });
            }
            // Queue the single basket metrics update
            const workerQueues = new WorkerQueues();
            await workerQueues.updateSingleBasketMetrics(basket_id);
            res.send({
                success: true,
                message: "Basket metrics update queued successfully",
                data: {
                    basketId: basket_id,
                    basketName: basket.name
                }
            });
        } catch (err: any) {
            console.error(err);
            res.status(500).send({
                success: false,
                error: err.message || "Internal server error"
```

2.9.2 Impact

External attackers can enqueue repeated metrics jobs, causing CPU/RPC/DB exhaustion

2.9.3 Recommendation

1. Require authentication (and ideally admin authorization) for metrics triggers
2. Add rate limiting + request throttling for any “expensive” endpoints

2.10 Wallet_address Parameter Not Bound To Authenticated User

Several authenticated “position tracking” endpoints accept a caller-supplied wallet_address and pass it into core tracking logic without verifying that it matches the authenticated user’s wallet. This enables a user to request portfolio/position calculations for arbitrary wallets and to **persist** those results under their own user record via /v1/sync.

2.10.1 Code location

```
15:35:src/application/api/controllers/position_tracking.ts
    get_user_position: async (req: FastifyRequest, res: FastifyReply) => {
        try {
            const { _id }: any = req.user;
            const { basket_id }: any = req.params;
            const { wallet_address, live }: any = req.query;
            if (!wallet_address) {
                return res.status(400).send({
                    success: false,
                    message: "Wallet address is required"
                });
            }
            const position = await positionTrackingService.getUserPositionMetrics(
                _id,
                basket_id,
                wallet_address,
                { live: String(live).toLowerCase() === 'true' }

262:277:src/application/api/controllers/position_tracking.ts
    sync_user_positions: async (req: FastifyRequest, res: FastifyReply) => {
        try {
            const { _id }: any = req.user;
            const { wallet_address }: any = req.body;
            if (!wallet_address) {
                return res.status(400).send({
                    success: false,
                    message: "Wallet address is required"
                });
            }
            await positionTrackingService.syncUserPositionsFromWallet(
                _id,
                wallet_address
258:315:src/core/positions/position_tracking.ts
async getUserPositionMetrics(
    userId: string,
    basketId: string,
    walletAddress: string,
    options?: { live?: boolean }
): Promise<UserPositionMetrics | null> {
    // ...
    if (!positionTracking) {
        // Create new position tracking record
        positionTracking = await this.createPositionTrackingRecord(
            userId,
            basketId,
            walletAddress
```

2.10.2 Impact

- Users can request “portfolio” calculations for arbitrary wallets (privacy/information boundary issue)
- /v1/sync can persist arbitrary wallet-derived balances under the caller’s user record (data integrity risk)
- If any downstream logic relies on user_position_tracking for decisions (feature access, rewards, support), this can be abused

2.10.3 Recommendation

1. Remove the wallet_address parameter from these endpoints and always use the authenticated wallet (req.user.address or DB-backed user wallet)
2. If multi-wallet support is required, implement explicit wallet linking and ownership proof, and authorize access per linked wallet
3. Add input validation for wallet_address (base58 length/pattern) and bound live/days/limit parameters

2.11 Unbounded Mints List In Batch Balance Endpoint Enables RPC Amplification (DoS)

GET /v1/misc/batch_token_balances accepts an unbounded mints list (comma-separated string or array) and passes it into a batch RPC call that fetches **2×N** accounts (ATAs + mint accounts) in a single getMultipleAccountsInfo request. There is no max length enforcement, making the endpoint susceptible to resource exhaustion and log amplification.

2.11.1 Code location

```
33:53:src/application/api/controllers/misc.ts
    async get_batch_token_balances(req: FastifyRequest, res: FastifyReply) {
        try {
            const { mints }: any = req.query;
            const { address } = req.user as { address: string };

            // Parse mints - can be comma-separated string or array
            let mintArray: string[];
            if (typeof mints === 'string') {
                mintArray = mints.split(',').map((m: string) => m.trim()).filter
((m: string) => m);
            } else if (Array.isArray(mints)) {
                mintArray = mints;
            } else {
                return res.status(400).send({ error:
            }
            if (mintArray.length === 0) {
                return res.status(400).send({ error: "At least one mint address is required"
 });
            }
            const balances = await getBatchTokenBalances(mintArray, address);
            return res.status(200).send({ success: true, data: { balances } });
        } catch (err: any) {
            console.error(err);
            return res.status(500).send({ error: err.message || "Internal server error" });

95:115:src/infrastructure/solana/connection.ts
export const getBatchTokenBalances = async (
    mints: string[],
    user: string
): Promise<Record<string, number>> => {
    // ...
    const atas = mints.map(mint =>
        getAssociatedTokenAddressSync(new PublicKey(mint), new PublicKey(user))
    );
    const mintPubkeys = mints.map(mint => new PublicKey(mint));
    const allAccountsToFetch = [...atas, ...mintPubkeys];
    const allAccountInfos = await connection.getMultipleAccountsInfo(allAccountsToFetch);
```

2.11.2 Impact

- Excessive CPU and memory use from parsing very large lists of mints
- RPC amplification via oversized getMultipleAccountsInfo calls (errors/retries/log noise)
- Potential denial of service against the API and/or upstream Solana RPC provider

2.11.3 Recommendation

1. Enforce a hard maximum length for mints (e.g., 20–50) and reject larger requests
2. Validate each mint as a base58 Solana public key (length/pattern) before processing
3. Chunk getMultipleAccountsInfo requests (respect RPC limits) and cap total work per request
4. Add per-user rate limiting for the batch endpoint

2.12 Unbounded Room Subscription Via `Subscribe:Prices` Enables WebSocket Memory/CPU DoS

Authenticated clients can request subscription to arbitrary “price rooms” with an unbounded list of token strings. There are no limits on list length or token length, allowing a single authenticated user to create/join huge numbers of rooms and inflate server memory/CPU overhead.

2.12.1 Code location

```
72:80:src/shared/utils/socket_utils.ts
// Example: Handle price updates subscription (optional - prices are broadcast to all)
socket.on('subscribe:prices', (data: any) => {
    if (data?.tokens && Array.isArray(data.tokens)) {
        data.tokens.forEach((token: string) => {
            socket.join(`price:${token}`);
        });
        console.log(`User ${userId} subscribed to price updates for tokens:`, data.tokens);
    }
});
```

2.12.2 Impact

- Memory/CPU exhaustion from pathological room joins (authenticated DoS)
- Log amplification (tokens list is logged)

2.12.3 Recommendation

1. Enforce strict limits (e.g., max 20–50 tokens per request; max token string length)
2. Validate tokens against base58 format and/or a server-side allowlist of supported mints
3. Rate-limit “subscribe” events per socket/user

2.13 Unbounded Async Log Processing Enables On-Chain Event Flood DoS

The listener processes each logsNotification by spawning an asynchronous handleLogs(...) task without any backpressure or concurrency limits. Each task performs multiple RPC calls (e.g., getSignatureStatus, getParsedTransaction, getTransaction) and can be CPU/memory intensive. An attacker can spam transactions that touch the program to generate a high volume of notifications, causing the listener to fall behind, exhaust memory/CPU, and/or trip RPC rate limits (which can cascade into missed or delayed off-chain processing).

2.13.1 Code location

```
199:244:src/application/listeners/services/program_event_listener.ts
// Handle log notifications
if (response.method === 'logsNotification' && response.params) {
    // ...
    // Process logs asynchronously
    this.handleLogs(logsData, context).catch((error) => {
        console.error(`[ProgramEventListener] Error processing logs for ${signature}:`, error);
    });
}

278:337:src/application/listeners/services/program_event_listener.ts
private async handleLogs(logs: Logs, context: any): Promise<void> {
    // ...
    const status = await this.connection.getSignatureStatus(signature);
    // ...
    const event = await this.parser.parseTransactionFromLogs(logs, signature, slot);
    // ...
}

28:90:src/application/listeners/services/parser/transaction_parser.ts
// Fetch parsed transaction (handles lookup tables automatically)
const parsedTransaction = await this.connection.getParsedTransaction(signature, {
  commitment: "confirmed",
  maxSupportedTransactionVersion: 0,
}).catch(() => null);
// Fallback to raw transaction
return this.parseFromRawTransaction(signature, slot);
```

2.13.2 Impact

- Listener CPU/memory exhaustion or RPC quota exhaustion (DoS)
- Delayed or missed processing of deposits/withdrawals and other program events

2.13.3 Recommendation

1. Add backpressure: queue incoming signatures and process with bounded concurrency (e.g., p-queue/p-limit and a max queue size
2. Prefer routing all events into BullMQ with a controlled worker pool rather than spawning unbounded async tasks
3. Add rate limiting / shedding: drop or defer processing when behind, and emit health alerts
4. Reduce per-event RPC calls (cache, batch, or skip confirmation checks when safe)

2.14 Token Program ID Fallback Risks Fund Misdirection

When the token program ID lookup fails (RPC/metadata outage), the system defaults to TOKEN_PROGRAM_ID instead of failing. This can cause incorrect account derivations for Token-2022 mints.

2.14.1 Code location

```
285:291:src/core/diversifi_service.ts
} catch (error: any) {
    console.error("Error getting token program id. Defaulting to TOKEN_PROGRAM_ID");
    const defaultProgramId = TOKEN_PROGRAM_ID;
    this.cacheManager.set(CacheType.TOKEN_PROGRAM_ID, tokenKey, defaultProgramId);
    return defaultProgramId;
}
```

2.14.2 Impact

If a Token-2022 mint is incorrectly treated as SPL Token, vault/account derivations will be wrong. Transactions may fail or, in worst case, funds could be sent to unexpected accounts.

2.14.3 Recommendation

1. Fail closed: throw an error instead of defaulting
2. Implement retry with exponential backoff before failing
3. Cache negative lookups for short TTL but never guess the program ID
4. Add monitoring/alerting for lookup failures

2.15 Slippage Default Lacks Upper Bounds

Swap slippage defaults to 100 bps (1%) and retry strategies may increase slippage without enforced caps. This can result in trades executing at materially worse prices.

2.15.1 Code location

```
108:116:src/core/diversifi_service.ts
const effectiveSlippage = slippageBps ?? 100; // Default: 100 bps (1%)
```

2.15.2 Impact

During market volatility, retry strategies could expand slippage beyond user intent, causing financial harm through poor execution prices.

2.15.3 Recommendation

1. Enforce hard upper slippage caps per basket and per-user intent
2. Log slippage used per swap for audit trail
3. Abort transactions if retry would exceed maximum tolerance

2.16 Token Accepted Via Query Parameter

JWT tokens can be passed via query parameter (`?token=...`), which exposes them in URLs that are logged, cached, and shared.

2.16.1 Code location

```
60:76:src/infrastructure/security/jwt/jwt_auth.ts
// Try query parameter
if (source.query?.token) {
    return source.query.token;
}
```

2.16.2 Impact

Tokens in URLs leak via server logs, browser history, referrer headers, analytics, and shared links.

2.16.3 Recommendation

1. Remove query parameter token support
2. Require tokens only via `Authorization: Bearer` header
3. If URL tokens are necessary for specific flows, use short-lived, single-use codes

2.17 Signature And Amount Precision Loss

User-provided amounts are converted through JavaScript Number type before scaling, which can introduce floating-point precision errors for large values.

2.17.1 Code location

```
347:351:src/application/api/controllers/positions.ts
new BN(Math.floor(Number(amount) * 10 ** usdc_decimals))
```

2.17.2 Impact

Large decimal inputs may be incorrectly converted, resulting in users receiving slightly more or less than intended.

2.17.3 Recommendation

1. Parse amounts as decimal strings
2. Use a precise decimal library (big.js, decimal.js) for scaling
3. Convert to BN using string arithmetic, not floating-point

2.18 No Payload Validation Or Dead-Letter Queue For Background Jobs

All background job queues (positions, transaction history, program events) lack payload validation and dead-letter queues. Workers directly trust job.data without validation. After retry attempts are exhausted, failed jobs are silently dropped with no record or mechanism to analyze/reprocess.

2.18.1 Code location

```
Event queue adds jobs without validation:
23:47:src/application/listeners/event_queue.ts
this.queue.add("processEvent", {...}, { removeOnComplete: true, attempts: 3, backoff:
{exponential} })


Workers trust job.data directly:
324:338:src/application/jobs/index.ts
new Worker('transactionHistory' | 'positions' | 'programEvents', async (job) => {...})

12:57:src/application/jobs/workers/program_events.ts
// Reconstructs ProcessedEvent without validation
```

2.18.2 Impact

- Malformed or malicious payloads cause runtime errors
- Poison messages cause repeated worker crashes
- After exhausting retry attempts, jobs are silently dropped with no audit trail
- No mechanism to analyze or reprocess failed jobs
- Missing required fields, invalid signatures, or wrong types crash handlers

2.18.3 Recommendation

1. Validate and sanitize job payloads with schema validation (zod, ajv) before processing
2. Reject invalid jobs early with typed errors
3. Add a dedicated DLQ (separate queue) for failed jobs across all queues
4. Set a cap on DLQ size and alert on entries
5. Log structured error information for failed jobs
6. Add type definitions for all job payloads

2.19 Oracle Prices Accepted Without Freshness/Outlier Checks

Switchboard Crossbar prices are accepted with only basic type validation (finite numbers) and without freshness/outlier checks. Additionally, when decoding on-chain oracle quotes, the code does not validate recency (timestamp/slot), allowing stale data to be used.

2.19.1 Code location

```
192:209:src/infrastructure/oracles/switchboard/switchboard_service.ts
const validResults = results.filter((curr: any) => {
    const num = Number(curr);
    return curr !== null && curr !== undefined && !isNaN(num) && isFinite(num);
});

if (validResults.length === 0) {
    // Retry fetching this specific feed
    const retryPrice = await this.retryFetchSingleFeed(feedId, mint);
    return {
        mint,
        price: retryPrice
    };
}

const sum = validResults.reduce((acc: number, curr: any) => acc + Number(curr), 0);
const average = sum / validResults.length;

86:93:src/infrastructure/oracles/switchboard/switchboard_service.ts
const ix = await this.createEd25519Instructions([feedId]);
const quote = sb.OracleQuote.decode(ix.data);
const info = quote.feedInfos?.[0];
if (!info) return 0;
const val = Number(info.value.toString());
// Normalize from 18-decimal fixed-point to USD number
const normalized = val / 1e18;
```

2.19.2 Impact

- Stale or extreme outlier values can skew price averages
- May misprice TVL and index values if upstream filtering fails
- System may operate on outdated price information during oracle/RPC issues

2.19.3 Recommendation

1. Enforce >0 and reasonable range checks for prices
2. Implement outlier rejection (median, trimmed mean)
3. Require freshness (max age per feed) and verify timestamps/slots for both Crossbar responses and on-chain quote data (`info`)
4. Fail closed (or enter safe fallback) on stale/anomalous data; log + alert on repeated staleness

2.20 Weak TransactionGroupId Derivation Causes Collision Risk

The transactionGroupId used to associate init/execute/refund transactions is derived from only the first 8 characters of the signature. This creates collision risk where different transactions could be misassociated.

2.20.1 Code location

```
166:173:src/core/event_handlers/base_transaction_handler.ts
// Extract transactionGroupId from init signature (first 8 chars)
const transactionGroupId = event.signature.substring(0, 8);
```

2.20.2 Impact

- Different transactions with the same 8-character prefix could be misassociated
- Execute or refund events could be linked to wrong pending transactions
- Accounting and state integrity compromised

2.20.3 Recommendation

1. Use the full signature instead of truncation
2. Use a program-emitted unique memo/ID for correlation
3. Enforce strict equality on longer identifiers
4. Add collision detection and alerting

2.21 Refund Selection Uses FIFO Without PDA/GroupId Verification

Refund transaction selection queries by user, basket, and type with FIFO ordering but does not verify the PDA or transactionGroupId. This can cause refunds to be applied to the wrong pending transaction.

2.21.1 Code location

```
481:491:src/core/event_handlers/base_transaction_handler.ts
const transactionToRefund = await db.transaction_history.findOne({
    user: userDoc._id,
    basket: basket._id,
    transactionType,
    state: { $in: ['processing', 'partially_executed'] },
    refundedAt: null,
    executedAt: null
}).sort({ createdAt: 1 }); // Oldest first (FIFO)
```

2.21.2 Impact

- Multiple concurrent transactions for same user/basket may be mismatched
- Refund events could mark the wrong transaction as refunded
- Race conditions cause incorrect state transitions

2.21.3 Recommendation

1. Include PDA and transactionGroupId in refund selection query
2. Verify event accounts match the transaction's PDA
3. Reject ambiguous matches when multiple candidates exist
4. Add alerts for multi-candidate refund scenarios

2.22 Minimal Input Validation On Critical Parameters

Validation schemas accept arbitrary strings for addresses and signatures without format, length, or pattern checks.

2.22.1 Code location

```
13:17:src/application/api/validation/positions.ts
required: ['basket_address', 'amount'],
properties: {
    basket_address: { type: 'string' },
    amount: { type: 'number' }
}

7:9:src/application/api/validation/auth.ts
address: { type: 'string' }
```

2.22.2 Impact

Oversized or malformed inputs cause resource strain, noisy logs, and potential parsing errors.

2.22.3 Recommendation

1. Add base58 pattern validation for Solana addresses
2. Enforce length constraints (32-44 chars for addresses, 88 chars for base58 signatures)
3. Add min/max bounds for numeric amounts
4. Set maxLength on all string inputs

2.23 Public Endpoints Enable Wallet/Code Enumeration

Unauthenticated endpoints for validating access codes and checking wallet links can be used to enumerate valid codes and associated wallets.

2.23.1 Code location

```
23:37:src/application/api/routes/auth.ts
url: '/validate_code' ... // No authentication
url: '/check-wallet-link' ... // No authentication
```

2.23.2 Impact

Adversaries can scan to discover which access codes are valid and which wallets are linked.

2.23.3 Recommendation

1. Add rate limiting and tarpit responses
2. Consider CAPTCHA for public validation endpoints
3. Return consistent response times regardless of validity

2.24 Lookup Tables Allow Duplicate Address Rows

The lookup_tables collection indexes address but does not enforce uniqueness. Concurrent upserts can create multiple documents with the same address.

2.24.1 Code location

```
20:27:src/infrastructure/storage/database/models/lookup_tables.ts
LookupTableSchema.index({ address: 1 });
```

2.24.2 Impact

Duplicate entries for the same on-chain lookup table cause operational confusion and may lead to inconsistent behavior when fetching table data.

2.24.3 Recommendation

1. Add unique: true to the address index
2. Consider compound unique with adminAuthority + address if multiple authorities are expected
3. Add upsert logic to prevent race conditions

2.25 Infinite Recursion Bug In GetTokenPrice Function

The getTokenPrice function calls itself recursively without any termination condition, causing a guaranteed stack overflow crash if ever invoked.

2.25.1 Code location

```
20:23:src/infrastructure/solana/connection.ts
export const getTokenPrice = async (mint: string) => {
    const price: any = await getTokenPrice(mint); // Calls itself!
    return price;
};
```

2.25.2 Impact

- Stack overflow crash if function is called
- Denial of service if exposed through any code path
- Dead code that indicates incomplete implementation

2.25.3 Recommendation

1. Implement actual price fetching logic (e.g., from Jupiter, Pyth, or another oracle)
2. If the function is unused, remove it to prevent accidental invocation
3. Add unit tests to catch recursive bugs

2.26 Hardcoded JWT Secret Fallback Enables Token Forgery

The JWT verification logic includes a hardcoded fallback secret that is used when the JWT_SECRET environment variable is not set. This same default value is documented in the environment template, making it publicly known.

2.26.1 Code location

```
31:35:src/infrastructure/security/jwt/jwt_auth.ts
const decoded = jwt.verify(cleanToken, process.env.JWT_SECRET ||
 'super-secret-solutiofi-jwt-key') as JWTPayload;
if (!decoded.address) {
    return { success: false, error: "Invalid token" };
}

54:56:env.template
# JWT secret for authentication (use a strong random string in production)
JWT_SECRET=super-secret-solutiofi-jwt-key
```

2.26.2 Impact

If JWT_SECRET is unset in any environment, attackers can forge valid JWTs for arbitrary users using the known default secret, enabling complete authentication bypass.

2.26.3 Recommendation

1. Remove the fallback entirely; fail fast if JWT_SECRET is missing
2. Replace the template default with an empty placeholder and mandatory comments
3. Use a cryptographically random 256-bit secret per environment
4. Store secrets via secret manager and validate presence at startup

2.27 Debug Backdoor Bypasses Signature Verification

A debug variable test exists in the authentication controller. When truthy, it completely bypasses wallet signature verification, allowing any address/signature combination to authenticate.

2.27.1 Code location

```
5:5:src/application/api/controllers/auth.ts
let test = 0;

31:33:src/application/api/controllers/auth.ts
const isValid = test || await verifySignature(address, signature);
if (!isValid) return res.status(401).send({ error: "Invalid signature" });
```

2.27.2 Impact

If test is accidentally set to truthy (via code change, hot reload, or merge conflict), all signature verification is disabled. Attackers can authenticate as any wallet address without proof of key ownership.

2.27.3 Recommendation

1. Remove the `test` variable and bypass logic entirely
2. If feature flags are needed, use environment-based flags with fail-secure defaults
3. Add unit tests to ensure signature verification cannot be bypassed

2.28 Static Admin Access Code Via Query Parameter

Admin endpoints use a shared static "access code" passed as a query parameter for authorization. The default code is ADMIN123 and is documented in the environment template.

2.28.1 Code location

```
319:324:src/application/api/controllers/auth.ts
// Validate admin code
if (!code || code !== admin_access_code) {
    return res.status(401).send({
        error: "Unauthorized: Invalid admin access code"
    });
}

57:60:env.template
# Admin access code for accessing access codes endpoint
# Default: ADMIN123 (change in production!)
ADMIN_ACCESS_CODE=ADMIN123
```

2.28.2 Impact

- Query parameters are logged by proxies, WAFs, and analytics
- The default code is guessable and publicly documented
- No rate limiting enables brute-force attacks
- Successful exploitation grants access to sensitive access code listings

2.28.3 Recommendation

1. Replace with JWT-based role authentication (admin role claim)
2. Pass secrets via Authorization header, never query parameters
3. Implement aggressive rate limiting on admin endpoints
4. Use high-entropy secrets rotated frequently if temporary codes are necessary

2.29 Jupiter API Key Logged To Console

The Jupiter client logs the API key directly to stdout during initialization. This exposes the secret to anyone with access to application logs.

2.29.1 Code location

```
26:26:src/infrastructure/blockchain/jupiter/jupiter_client.ts
console.log('JUPITER_API_KEY', jupiter_api_key);
```

2.29.2 Impact

API key leaks to logs. Any log consumer (developers, operators, log aggregation services, attackers with log access) can exfiltrate and abuse the quota or impersonate API calls.

2.29.3 Recommendation

1. Remove secret logging entirely
2. Add runtime check to error if key is missing, but never print the value

Manual Testing

3 Manual Security Verification Test Cases

This document describes what security properties were checked and provides manual security test scenarios to evaluate them.

This is a security-oriented test plan (threat-driven), not a functional QA checklist.

3.1 Component Map

Auth (/v1/auth/):
- GET /v1/auth/nonce
- POST /v1/auth/verify
- GET /v1/auth/validate-code (public)
- GET /v1/auth/check-wallet-link-by-address (public)
- GET /v1/auth/check-wallet-link (authenticated)
- POST /v1/auth/verify-code (authenticated)
- GET /v1/auth/access-codes (public)
- POST /v1/auth/refresh (refresh token based)
- POST /v1/auth/logout (currently public by design)

Baskets (/v1/baskets/):
- GET /v1/baskets/:basket_id/performance
- POST /v1/baskets/:basket_id/update-metrics
- GET /v1/baskets/:basket_id

Positions (/v1/positions/, authenticated):
- GET /v1/positions/open/tx
- GET /v1/positions/open/confirm
- GET /v1/positions/close/tx
- GET /v1/positions/close/confirm
- GET /v1/positions/transactions
- GET /v1/positions/cancel/tx
- GET /v1/positions/cancel/confirm
- GET /v1/positions/cancel/clear
- GET /v1/positions/refund/check
- GET /v1/positions/refund/tx
- GET /v1/positions/refund/confirm

Misc (/v1/misc/, authenticated):
- GET /v1/misc/token_meta
- GET /v1/misc/token_balance
- GET /v1/misc/batch_token_balances

User (/v1/user/, authenticated)
- GET /v1/user/

Position tracking & portfolio (prefix /v1, authenticated):
- GET /v1/position/:basket_id
- GET /v1/portfolio
- POST /v1/sync
- GET /v1/position/:basket_id/performance
- GET /v1/top-positions
- GET /v1/analytics

3.2 Debug Mode

- Examples (not exhaustive): GET /debug/connections, GET /debug/test-redis, POST /debug/process-transaction, POST /debug/inspect-transaction

(must be gated in production)

3.3 WebSockets

Namespaces: root / and authenticated /app

Client-emitted events (from bootstrapHandlers):

- subscribe:baskets, unsubscribe:baskets
- subscribe:prices, unsubscribe:prices (joins price:<token> rooms)
- ping → server replies pong
- get:user-info → server replies user-info

3.4 Queues (BullMQ / Redis)

Queues (from src/infrastructure/messaging/queues.ts):

- positions (job name: positions)
- transactionHistory (job names: updateTransactionHistory, markTransactionHistoryFailed)
- basketMetrics (job names: updateMetrics, updateSingleMetrics)
- websockets (job names include handlePositionUpdate, handleBasketUpdate, handlePriceUpdate, etc.)

3.5 Business Logic & User Funds Safety

3.5.1 Correct correlation across the “open → execute → confirm/refund” lifecycle

What is checked

- Each on-chain event is matched to the correct user, basket, and transaction instance.
- No ambiguous fallback behavior can bind an event to the “wrong pending record.”

Scenarios

- Concurrent operations: multiple open/close actions for the same user and basket within a short window.
- Ambiguity stress: multiple pending records exist for the same basket (and potentially same user).
- Failure injection: parsing failures, RPC timeouts, or missing metadata that could trigger fallback logic.

Secure expected outcome

- Correlation remains stable under concurrency.
- If correlation is ambiguous, the system fails safely (does not guess) and flags for reconciliation.

3.5.2 Idempotency & replay resistance (API + workers + listener)

What is checked

- Retried requests, repeated confirmations, and repeated worker jobs do not cause duplicated state transitions.

Scenarios

- Repeat confirmation/cancel/refund calls with the same identifiers.
- Restart jobs/workers mid-flight and observe whether actions re-run safely.
- Re-deliver the same event/job payload multiple times (queue replay).

Secure expected outcome

- Replays are deduplicated by stable idempotency keys.
- State transitions are monotonic and consistent with on-chain truth.

3.5.3 Numeric safety (amount precision and scaling)

What is checked

- Amounts are processed with exact arithmetic (no float precision loss).
- Boundaries exist for “tiny” and “huge” values.

Scenarios

- Very small decimal inputs (near dust).
- Large amounts near upper UI/business constraints.
- Inputs with many decimal places or scientific notation.

Secure expected outcome

- Consistent conversion and storage across API > tx > DB.
- Inputs outside policy bounds are rejected early.

3.5.4 Idempotency & replay resistance (API + workers + listener)

What is checked

- Retried requests, repeated confirmations, and repeated worker jobs do not cause duplicated state transitions.

Scenarios

- Repeat confirmation/cancel/refund calls with the same identifiers.
- Restart jobs/workers mid-flight and observe whether actions re-run safely.
- Re-deliver the same event/job payload multiple times (queue replay).

Secure expected outcome

- Replays are deduplicated by stable idempotency keys.
- State transitions are monotonic and consistent with on-chain truth.

3.6 Authentication (wallet + JWT) and session security

3.6.1 Nonce issuance (GET /v1/auth/nonce) and replay protection

What is checked

- Each on-chain event is matched to the correct user, basket, and transaction instance.
- No ambiguous fallback behavior can bind an event to the “wrong pending record.”

Scenarios

- Concurrent operations: multiple open/close actions for the same user and basket within a short window.
- Ambiguity stress: multiple pending records exist for the same basket (and potentially same user).
- Failure injection: parsing failures, RPC timeouts, or missing metadata that could trigger fallback logic.

Secure expected outcome

- Correlation remains stable under concurrency.
- If correlation is ambiguous, the system fails safely (does not guess) and flags for reconciliation.

3.6.2 Signature verification (POST /v1/auth/verify)

What is checked

- Signature verification cannot be bypassed and is strictly validated.

- Scenarios

- Invalid signature for a real address.
- Signature for a different message/nonce.
- Oversized or malformed signature strings (format/length/pattern).

- Secure expected outcome

- Invalid signatures are rejected deterministically with minimal leakage.

3.6.3 JWT verification and token transport

What is checked

- JWT verification is strict (algorithm, expiry, and parsing behavior).
- Tokens are accepted only through intended channels (prefer header over URL/query).

Scenarios

- Tampered tokens (header/payload changes).
- Expired tokens / missing expiry / future-not-before claims.
- Token passed via query parameters or unexpected sources.

Secure expected outcome

- Only properly issued tokens are accepted; unsafe transports are rejected.
- Tokens never appear in plaintext logs.

3.6.4 Refresh and logout semantics (POST /v1/auth/refresh, POST /v1/auth/logout)

What is checked

- Refresh tokens are rotation-safe and reuse-resistant.
- Logout only revokes the caller’s own sessions and cannot be abused for targeted revocation.

Scenarios

- Reuse the same refresh token after refresh.
- Attempt logout with a token that does not belong to the caller.
- Concurrent refresh calls (same token) to test race behavior.

Secure expected outcome

- Refresh tokens are rotated or reuse is detected.
- Logout is authorization-bound.

3.6.5 Public auth surfaces and enumeration resistance

Public endpoints include GET /v1/auth/validate-code, GET /v1/auth/check-wallet-link-by-address, and GET /v1/auth/access-codes.

What is checked

- Rate limiting / tarpit behavior, consistent errors, and absence of enumerability at scale.

- Scenarios

- High-rate probing with random and structured guesses.
- Timing analysis (valid vs invalid) and response-shape comparison.

- Secure expected outcome

- Consistent responses and throttling; no practical enumeration advantage.

3.7 Authorization (BOLA/IDOR) and identity binding

3.7.1 Wallet/user binding in portfolio/position tracking

What is checked

- Any endpoint that accepts a wallet-like parameter (query/body) must enforce that it is owned/linked to the authenticated user.

Scenarios

- Provide a third-party wallet address and request portfolio/position computations.
- Attempt to persist third-party wallet-derived results via sync endpoints.

Secure expected outcome

- Requests are rejected unless the wallet is explicitly linked and authorized.

3.7.2 Ownership checks for transaction confirmation and cancellation

What is checked

- Confirmation/cancel flows must be bound to the initiating user and the correct transaction instance.

- Scenarios

- Attempt to confirm/cancel using identifiers associated with a different user.
- Attempt to confirm/cancel with malformed or unrelated signatures.

- Secure expected outcome

- Only the owner can confirm/cancel; ambiguous matches fail safely.

3.8 Input validation, parsing robustness, and ambiguity handling

3.8.1 Schema strictness and bounds

What is checked

- All string inputs have reasonable bounds; Solana public keys/signatures are validated by format and length.
- List inputs have strict max sizes (e.g., token mint arrays).
- Numeric inputs are bounded and parsed safely.

- Scenarios

- Overlong strings (addresses/signatures/IDs).
- Unicode/control characters.
- Large arrays and deeply nested JSON.

- Secure expected outcome

- Invalid/oversized inputs are rejected at the boundary with 4xx responses.

3.8.2 Parameter pollution and ambiguous query handling

What is checked

- Duplicate parameters and mixed array/string inputs do not create ambiguous behavior.

- Scenarios

- Duplicate query keys (e.g., multiple limit values).
- Mixed encodings (comma-separated string vs repeated keys).

- Secure expected outcome

- Deterministic handling or explicit rejection of ambiguous requests.

3.9 Availability / DoS and cost-amplification resistance

3.9.1 Expensive API endpoints and job triggers

What is checked

- Endpoints that enqueue work or trigger heavy DB/RPC compute are authenticated, rate-limited, and deduplicated.

Scenarios

- Burst calls to POST /v1/baskets/:basket_id/update-metrics.
- Large queries to GET /v1/positions/transactions and similar list endpoints.

Secure expected outcome

- Rate limits and dedup prevent queue floods and costly repeated work.

3.9.2 RPC amplification

What is checked

- Any endpoint that maps user input into RPC batch calls is chunked and capped.

Scenarios

- Large mint lists to GET /v1/misc/batch_token_balances.

Secure expected outcome

- Request size caps, chunked RPC calls, and per-user throttles prevent upstream exhaustion.

3.9.3 WebSocket flooding and room explosion

What is checked

- Client-emitted events have rate limits and payload size limits.
- Room subscriptions are capped and validated.

Scenarios

- High-frequency event spam (subscribe:prices loops).
- Very large token lists and long token strings in subscribe:prices.

Secure expected outcome

- Server sheds load (rate-limits/disconnects) and bounds memory growth.

3.10 Web security posture (browser-facing risks)

3.10.1 CORS configuration

What is checked

- CORS origins are constrained per environment; credentials usage is safe.

Scenarios

- Cross-origin requests from untrusted origins to authenticated endpoints.
- Verify whether credentials are allowed and whether origin reflection occurs.

Secure expected outcome

- Only trusted origins are allowed; no permissive reflection with credentials.

3.10.2 HTTP method semantics and CSRF considerations

What is checked

- State-changing operations should not be reachable via GET, and CSRF is addressed if cookies are used.

Scenarios

- Review state-changing routes under /v1/positions/ that are GET.

Secure expected outcome

- State changes are POST/PUT/PATCH with CSRF protections as needed.

3.11 WebSockets authorization, isolation, and session lifecycle

3.11.1 Authentication sources and session expiry

What is checked

- Only intended token sources are accepted during WS handshake.
- Connected sockets lose access when tokens are revoked/expired.

Scenarios

- Connect with expired tokens, revoked tokens, or tokens sent via unintended handshake locations.
- Logout/refresh while sockets remain connected and observe continued event delivery.

Secure expected outcome

- Revocation and expiry are enforced for already-connected sessions.

3.11.2 Data isolation in server-emitted events

What is checked

- User-specific events are delivered only to the correct user:<id> room.
- Batch/bulk operations never leak cross-user payloads.

Scenarios

- Observe payloads for multi-user/bulk broadcast paths.
- Verify that user IDs and payloads are scoped to the intended recipient.

Secure expected outcome

- No cross-user data appears on unrelated clients.

3.11.3 Subscription/room mechanics

What is checked

- Rooms joined via subscribe:prices are bounded and validated against supported tokens/mints.

Scenarios

- Join large numbers of rooms; join rooms with invalid token strings.

Secure expected outcome

- Server enforces bounds and rejects invalid/unrecognized tokens.

3.12 Listener pipeline (Solana logs > parsing > routing) integrity

3.12.1 Backpressure and concurrency control

What is checked

- logsNotification handling does not spawn unbounded async work.
- The listener remains stable under high event volume.

Scenarios

- Burst program activity; observe listener memory/CPU and lag.

Secure expected outcome

- Bounded concurrency with queueing; graceful shedding under overload.

3.12.2 Deduplication correctness (seen vs successfully processed)

What is checked

- Deduplication keys represent successful processing, not merely “seen.”

Scenarios

- Force transient failures (RPC/DB timeouts) and verify events are retried.

Secure expected outcome

- Failures are retryable; events are not permanently dropped due to early marking.

3.12.3 Crash/restart recovery

What is checked

- Last-processed state tracking does not advance past unprocessed events.

- Scenarios

- Induce a failure during routing and restart; verify catch-up behavior.

- Secure expected outcome

- Missed signatures are replayed and processed after restart.

3.13 Background jobs/queues (BullMQ) security

3.13.1 Payload validation and type safety

What is checked

- Workers validate job.data shape/types before using it.

Scenarios

- Enqueue malformed payloads to each queue (positions, transactionHistory, basketMetrics, websockets).

Secure expected outcome

- Jobs are rejected safely, recorded for analysis (DLQ), and do not crash workers repeatedly.

3.13.2 Retries, backoff, DLQ, and failure visibility

What is checked

- Transient failures retry with backoff; persistent failures go to DLQ with actionable context.

- Scenarios

- Simulate transient dependency outages (DB/RPC) and observe job behavior.

- Secure expected outcome

- No silent drops; failures are observable and recoverable.

3.13.3 Job deduplication / replay and authorization binding

What is checked

- Duplicate jobs for the same logical action are coalesced (jobId/idempotency keys).
- Jobs cannot be used to emit cross-user WebSocket events.

- Scenarios

- Re-enqueue identical logical operations.
- Attempt to send a job payload referencing another user’s userId.

- Secure expected outcome

- Replays don’t cause repeated side effects; userId scoping is enforced.

3.14 External HTTP, SSRF, and untrusted URL ingestion

3.14.1 URL allowlisting and scheme restrictions

What is checked

- Only allowed schemes (typically HTTPS) are permitted.
- Unsafe schemes and internal targets are blocked.

Scenarios

- Metadata-driven URL fetches (Helius metadata/image, JSON URIs) and Arweave URL uploads.
- Redirect-based SSRF and DNS rebinding patterns.

Secure expected outcome

- Requests to internal/private/link-local/loopback targets are blocked.
- Redirects are disabled or re-validated after redirect.

3.14.2 HTTP client bounds and logging hygiene

What is checked

- Strict timeouts, body size limits, and conservative redirect policies.
- Logs do not print full third-party payloads or secrets.

Scenarios

- Slow responses, huge bodies, and malformed JSON responses.
- Untrusted metadata containing large strings or control characters.

Secure expected outcome

- Bounded resource use; safe logging (truncate/sanitize) with no secret leakage.

3.15 Secrets management and sensitive logging review

3.15.1 Secrets management and sensitive logging review

What is checked

- No secrets committed to source control (keys, service account JSON, etc.).
- No secrets are logged (API keys, JWTs, signatures, raw transactions).
- Debug endpoints mask secrets and are disabled in production.

Scenarios

- Startup log review under realistic env vars.
- Exercise endpoints that touch external integrations and inspect logs.
- Validate /debug/ behavior under production-mode toggles.

Secure expected outcome

- Secrets are never printed; debug tooling is not internet-reachable in production.

3.16 Data layer integrity (MongoDB) and injection resistance

3.16.1 Uniqueness and integrity constraints

What is checked

- Unique indexes exist where required (e.g., transaction identifiers, lookup-table addresses).

Scenarios

- Concurrent inserts/upserts of the same logical record.

Secure expected outcome

- DB-level constraints prevent duplicates; application handles conflicts safely.

3.16.2 NoSQL operator injection resistance

What is checked

- User input cannot inject Mongo operators (e.g., $ne, $gt) into query filters.

Scenarios

- Inputs shaped like objects where strings are expected (where applicable).

Secure expected outcome

- Strict schema validation prevents operators from reaching DB queries.

IRONNODE

ironnode.io

Thank you for choosing us!