IRONNODE

ironnode.io



# MY CRYPTO CASINO

## dApp Security Audit

by **IronNode** from **Jan 09 - Jan 25, 2025**

# Executive overview

# 1 Executive Overview

## 1.1 Introduction

**MyCryptoCasino** is a Solana-based crash game and USDC casino vault implemented as a hybrid on-chain/off-chain system. On-chain, the solana-mycryptocasino Anchor program manages the casino's USDC bankroll, crash-round lifecycle (round IDs, statuses, caps), user/bet accounts, and referral accounting (including multi-level commission distribution). Off-chain, the bitrush-game-api NestJS backend provides the product-facing API and real-time client updates, orchestrates game flow, and continuously reconciles off-chain records against on-chain activity. Core infrastructure includes PostgreSQL (TypeORM) for persistence (users, games, transactions, referral metadata), Redis for caching and Socket.IO scale-out, and Solana RPC/WebSocket endpoints for transaction submission and program monitoring.

At a high level:

- A user authenticates via Sign-In With Solana (SIWS) (wallet challenge + signature) and receives JWT-based session credentials bound to their wallet address.
- The backend builds base64-serialized Solana transactions for user-signed actions (e.g., create user w/ referral, deposit, withdraw, referral NFT/rewards flows), verifies the resulting signatures on submission, and relays the transactions to Solana.
- The backend also performs operator-controlled actions using a privileged Solana authority (e.g., starting rounds, starting the game to snapshot an on-chain blockhash, revealing the secret to finalize the crash, and executing settlement/payout flows).
- A chain-sync pipeline (scheduled signature scanning plus optional real-time log subscriptions) decodes program activity using the Anchor IDL, routes events through a concurrency-limited processor with deduplication/checkpointing, updates Postgres state, and emits Socket.IO updates (game status/curve updates and transaction notifications) to connected clients.

This architecture supports a responsive casino experience while anchoring critical outcomes on-chain, but it also introduces explicit trust boundaries: privileged key custody and authorization for operator-signed instructions; semi-trusted inputs from HTTP/WebSocket clients and user-submitted signed transactions; and asynchronous reconciliation surfaces driven by Solana RPC/log streams and scheduled processing. Accordingly, the audit emphasizes correct wallet/session binding, strict authorization around operator actions, robust transaction/signature handling, provably-fair secret/commit–reveal correctness, and resilient event processing (idempotency, deduplication, and safe state transitions).

## 1.2 Findings and Remediations

During the  audit, we identified multiple security issues across severities (high, medium and low) 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 **MyCryptoCasino** team.

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

## 1.3 Audit Objectives

- **API Security and Authentication**
  - Wallet authentication (SIWS): nonce issuance/expiry/consumption, signature verification, and replay resistance.
  - JWT/session lifecycle: token issuance, refresh flow (rotation/revocation), logout semantics, and cookie/session handling.
  - Authorization & abuse prevention: strict binding of userId ↔ walletAddress, privileged/admin endpoint controls, rate limiting (HTTP + WS), input validation/bounds, and safe error handling.
- **Solana Transaction Execution and User Funds Safety**
  - Transaction build/submit flows: safe unsigned-TX construction for user signing, strict signature verification on submission, and constrained admin co-signing only where required.
  - Key custody & privilege separation: protection of operator/admin keys used for game flow and settlement; separation of read-only RPC paths vs write-capable execution paths.
  - Execution safety: preflight/simulation/confirmation policy, retry limits, token mint/ATA correctness (USDC decimals), and robust failure/refund handling.
  - Idempotency & correlation: prevention of duplicate/replayed submissions and strict mapping between DB Transaction records and on-chain signatures/events.
- **Listener Pipeline, Event Integrity, and Reconciliation**
  - Program monitoring: correctness of historical signature scanning and real-time log subscriptions.
  - Parsing correctness & resilience: IDL-based instruction/event decoding, account mapping, and robustness against malformed/oversized logs or unexpected instruction variants.
  - Back pressure/dedup/reprocessing: bounded concurrency, dedup semantics that avoid permanent event loss, and checkpointing/retry behavior that preserves reprocessing guarantees on failure.
- **Asynchronous Processing (Schedulers/Queues) and DoS Resilience**
  - Cron/state-machine safety: resilience of game-engine loops, pending-transaction checks, batching, and any internal queues.
  - Cost-amplification defenses: protecting heavy RPC/DB work from untrusted triggers, bounding retries/backoff, and ensuring failures are observable/auditable.
  - Consistency guarantees: atomic updates where required and safe recovery paths to prevent partial off-chain state and desyncs.
- **Data Layer, Caching, WebSockets, and External Integrations**
  - PostgreSQL/TypeORM integrity: authorization at query boundaries (BOLA prevention), uniqueness/ consistency constraints, and safe persistence of user/transaction state.
  - Redid + WebSockets security: authentication, room/subscription controls, throttling, message validation, and prevention of cross-user data leakage.
  - External dependencies: Solana RPC/WS providers and NFT/referral ownership logic, plus any outbound services (observability, email, etc.), with attention to secrets/PII redaction and untrusted-input handling.

## 1.4 Scope

**Audit Scope:**

- Repository: bitrush-game-api
- Commit hash: 7f51f6d8c768a74beef647ab322a4acebcc82f11

## 1.6 Assessment Summary & Findings Overview

| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|
| 0 | 3 | 8 | 2 | 0 |

| Security analysis | Risk Level | Remediation Status |
|---|---|---|
| GitHub Packages token in .npmrc | HIGH | FIXED |
| Dockerfile copies .npmrc into runtime | HIGH | FIXED |
| Failed on-chain transactions can be processed as “successful” events | HIGH | FIXED |
| Referral commission dedup bug: “depth” checked vs stored is inconsistent | MEDIUM | FIXED |
| High-privilege CLI commands are present and can mutate DB / submit on-chain tx | MEDIUM | FIXED |
| Referral commission parsing uses empty-string fallback for referral owner | MEDIUM | FIXED |
| Transaction intent bypass: POST /transactions/submit-user-signed is not bound to the DB transactionId | MEDIUM | FIXED |
| Refresh tokens can be used as Bearer access tokens | MEDIUM | FIXED |
| Logout does not revoke refresh tokens server-side due to refresh cookie Path scoping | MEDIUM | FIXED |

| Security analysis | Risk Level | Remediation Status |
| --- | --- | --- |
| Global validation does not whitelist/strip unknown fields | MEDIUM | FIXED |
| CORS defaults to * with credentials: true | MEDIUM | FIXED |
| Floating-point math used for u64 token amounts/multipliers | LOW | FIXED |
| Unbounded Limit on network list queries | LOW | FIXED |

EXECUTIVE OVERVIEW

## 1.7 Remediation Notes

We reviewed the client's remediation changes on bitrush-game-api develop @ d3734d60beaef3b2bcb682cc94e6fe7ee722b0b8 and re-validated the issues via static code review. At this commit, 13/13 appear fixed in code.

- **GitHub Packages token in .npmrc (Fixed):** The hardcoded GitHub Packages token was removed from source control; .npmrc now references an environment-provided token (${NODE_AUTH_TOKEN}), eliminating the committed secret. Ops follow-through: token rotation + git history purge must still be done outside code.

- **Dockerfile copies .npmrc into runtime (Fixed):** The Dockerfile no longer copies .npmrc into the runtime stage; it uses BuildKit secret mounts to create a temporary .npmrc only during dependency install and deletes it afterward, preventing token persistence in built images.

- **Failed on-chain tx processed as "successful" events (Fixed):** Transaction parsing now fails closed when meta.err is present (no event emission for failed transactions), preventing DB reconciliation from treating reverted on-chain actions as successful.

- **Referral commission dedup "depth" mismatch (Fixed):** Commission records now use a consistent depth definition and update/merge existing rows instead of creating duplicates on retries/reprocessing.

- **High-privilege CLI commands (Fixed):** High-impact commands are now dev/test gated (only registered when ENVIRONMENT is development|test), reducing post-exploitation/insider amplification in production environments.

- **Referral commission empty-string referral owner fallback (Fixed):** Commission parsing no longer falls back to "" for referral owner; it fails parsing if ownerWallet can't be resolved, preventing creation of junk/invalid users/commission chains.

- **Transaction intent bypass (submit-user-signed) (Fixed):** The backend now stores an expected transaction message hash at build-time and verifies the submitted transaction matches it at submit-time, preventing instruction/account/amount swapping under the same transactionId.

- **Refresh tokens usable as Bearer access tokens (Fixed):** Access vs refresh tokens are now separated by a required typ claim and validated with separate secrets; the JWT strategy rejects non-access token types.

- **Logout refresh revocation blocked by cookie Path (Fixed):** Refresh cookie scope was broadened to /auth, ensuring /auth/logout receives it; logout now verifies refresh tokens properly while remaining idempotent.

- **Global validation whitelist/strip unknown fields (Fixed):** Global validation now enables whitelist + forbidNonWhitelisted + transform, closing common mass-assignment/extra-field injection paths.

- **CORS default * with credentials (Fixed as a misconfig failure mode):** CORS origins are now explicitly env-driven/validated (no silent fallback when unset). Ops note: production should still use an allowlist (not "*").

- **Floating-point math for u64 amounts/multipliers (Fixed):** Critical conversions now use BigNumber/BN conversion helpers (e.g., USDC base units + multiplier BPS), removing Math.floor(amount * scale) style float handling in Solana transaction construction.

- **Unbounded network list limit (Fixed):** Network paging DTOs now enforce a hard maximum (e.g., @Max(100)), preventing large limit values from driving expensive queries/responses.

# Findings

# 2 Findings

## 2.1 GitHub Packages Auth Token Committed In .Npmrc

The repository contains a hardcoded GitHub Packages _authToken in .npmrc. This is a direct secret leak. Anyone with repo access (or leaked source) can potentially use this token to authenticate to npm.pkg.github.com under your org namespace, enabling package tampering (supply-chain compromise) and/or unauthorized package reads, depending on token scopes.

### 2.1.1 Code location

- File Location: Npmrc lines 1-5

```
git-checks=false

@()-game:registry=https://npm.pkg.github.com
//npm.pkg.github.com/:_authToken="_REDACTED_"
```

### 2.1.2 Impact

If the token has write:packages (or broader repo scopes), an attacker can publish a malicious version of an internal dependency (e.g., @()-game/bitrush-commons) and compromise the API at build/run time. Even read-only scopes can leak proprietary packages.

### 2.1.3 Recommendation

To mitigate this issue, it is recommended to:

- Immediately revoke/rotate the token in GitHub (assume it is compromised).
- Remove the secret from the repository history (not just the tip) using git filter-repo/BFG, and invalidate old builds/images that might contain it.
- Stop committing .npmrc secrets: use CI secrets (NODE_AUTH_TOKEN) or BuildKit secrets for installs.
- Use least-privilege tokens (prefer read:packages for CI installs; separate write:packages to publishing workflows; no repo scope unless absolutely required).

## 2.2 Dockerfile Copies .Npmrc Into The Runtime Image

The production stage copies .npmrc into the final image. Since your .npmrc contains a GitHub auth token, the runtime container/image will carry that secret, expanding the blast radius to image registries, artifact scanners, and anyone with container read access.

### 2.2.1 Code location

- File Location: Dockerfile lines 1-14

```
FROM node:20-alpine3.22 AS build
WORKDIR /app
COPY package.json yarn.lock .npmrc ./
RUN yarn install
COPY . .
RUN yarn run build

FROM node:20-alpine3.22 as serve
WORKDIR /app
COPY --from=build /app/dist ./dist
COPY --from=build /app/package.json /app/yarn.lock /app/.npmrc ./
RUN yarn install --production
EXPOSE 3000
CMD ["yarn", "run", "start:prod"]
```

### 2.2.2 Impact

Secret exfiltration from the built image enables supply-chain compromise even if the Git repo is later cleaned.

### 2.2.3 Recommendation

To mitigate this issue, it is recommended to:

- Do not copy .npmrc into the runtime stage. Install deps in build stage and copy only node_modules + dist, or use BuildKit --mount=type=secret during install.
- Rebuild and redeploy after rotation; purge old images from registries.

## 2.3 Failed On-Chain Transactions Can Be Processed As “Successful” Events

SolanaRpcService.parseTransaction() parses instructions and emits events without checking whether the transaction failed on-chain (transaction.meta.err). This is extremely high risk because several events (notably deposit/withdraw) are derived from instruction args + accounts, not from final state deltas. A transaction that fails (insufficient funds, invalid accounts, wrong mint, etc.) can still be interpreted as a valid event and drive DB reconciliation.

### 2.3.1 Code location

- File Location: solana-rpc.service.ts lines 186-193

```
// Extract basic transaction information
const { meta, transaction: txData, slot, blockTime } = transaction;

if (!meta || !txData || !blockTime || txData.message?.instructions?.length === 0) {
   this.logger.warn(`Incomplete transaction data for ${signature}`);
   return { status: TransactionProcessingStatus.INCOMPLETE_DATA };
}
```

### 2.3.2 Impact

- Backend may update DB as if funds moved / bets happened when the transaction actually failed and was rolled back.
- This can create a DB/on-chain desync that can be exploited for financial gain depending on what downstream logic trusts (transaction history, “confirmed” state, balance monitoring, withdrawal eligibility, affiliate payouts, etc.).

### 2.3.3 Recommendation

To mitigate this issue, it is recommended to:

- Fail closed: in parseTransaction(), if meta.err is set, return a dedicated status (e.g., FAILED_ON_CHAIN) and do not emit any events.
- In historical sync, skip signatures with err (or process them through a “failed tx” path that only marks DB PENDING txs as FAILED).
- Optionally add stronger integrity: for financial instructions, validate effects using meta.preTokenBalances/postTokenBalances or account balance deltas (and only emit deposit/withdraw events when the token deltas match the instruction).

## 2.4 Referral Commission Dedup Bug: “Depth” Checked Vs Stored Is Inconsistent

Commission dedup checks depth: commissionLog.depth but stores commission.depth = commissionLog.depth + 1. This mismatch can lead to duplicate commission rows for the same bet/ancestor/descendant level (especially on reprocessing/retries), violating the “duplicate handling” audit goal.

### 2.4.1 Code location

- File Location: referral-commission.helper.ts lines 72-99

```
const existingCommission = await queryRunner.manager.findOne(ReferralCommission, {
   where: {
       betId: bet.id,
       ancestorId: ancestorUser.id,
       descendantId: descendantUser.id,
       depth: commissionLog.depth,
   },
});
if (existingCommission) {
   this.logger.log(`ℹ️ Commission already exists for level ${commissionLog.depth}, skipping`);
   continue;
}
// ...
commission.depth = commissionLog.depth + 1;
```

### 2.4.2 Impact

Duplicate referral commissions can be recorded and potentially paid/credited multiple times depending on downstream logic.

### 2.4.3 Recommendation

To mitigate this issue, it is recommended to:

- Use a single consistent definition of “depth” (either store commissionLog.depth as-is, or always store +1, but dedup with the same value).
- Add a DB-level unique constraint for (betId, ancestorId, descendantId, depth) and make inserts idempotent (ON CONFLICT DO NOTHING / catch unique violation).

## 2.5 High-Privilege CLI Commands Are Present And Can Mutate DB / Submit On-Chain Tx

The repo includes a CLI (src/cli.ts) and commands that can parse arbitrary transactions and optionally save them to DB, and can send privileged test transactions. These are not HTTP endpoints, but if an insider or attacker gets container exec / RCE, having these tools co-located increases the chance of fast, high-impact misuse.

### 2.5.1 Code location

- File Location: cli.ts lines 5-20

```
async function bootstrap() {
   config({ path: ".env" });
   process.env.CLI_MODE = "true";
   // ...
   const app = await CommandFactory.runWithoutClosing(CommandsModule, [
"warn", "error", "log"]);
```

### 2.5.2 Impact

Accelerates insider/RCE impact (forced DB reconciliation, on-chain submissions via internal services, etc.).

### 2.5.3 Recommendation

To mitigate this issue, it is recommended to:

- Build and ship a separate image/artifact for CLI tooling (not deployed alongside the API).
- Require explicit runtime deny/allow checks (e.g., CLI refuses to run unless ALLOW_CLI_IN_THIS_ENV=true).
- Ensure production containers are non-interactive (no exec) and follow least privilege.

## 2.6 Referral Commission Parsing Uses Empty-String Fallback For Referral Owner

In payout parsing, if referral owner resolution fails, the code sets referralAddress to "". Later, the commission pipeline treats that as an address and can create users/affiliate codes/commissions tied to it.

### 2.6.1 Code location

- File Location: cli.ts lines 5-20

```
return {
   playerAddress: await this.resolveUserWalletAddress(c.data.player),
   referralAddress: referralData?.ownerWallet || "",
   nftAddress: referralData?.nftAddress || "",
   referralCode: referralData?.code || "",
   amount: amountBN,
   commissionRate: rate,
   depth: c.data.level,
};
```

### 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 Intent Bypass: POST /Transactions/Submit-User-Signed Is Not Bound To The DB TransactionId

The submit endpoint is unauthenticated and only checks that the wallet in DB (Transaction.from) signed the submitted base64 transaction. It does not validate that the transaction instructions match what the backend originally built/stored for that transactionId/type (program ID, accounts, amounts, discriminator, etc.).

### 2.7.1 Code location

- File Location: transaction.controller.ts lines 192-214

```
@Post("submit-user-signed")
@CriticalThrottle()
async submitSignedTransaction(
    @Body() dto: SubmitSignedTransactionDto,
): Promise<GenericResponseDto<SubmitSignedTransactionResponseDto>> {
    const result = await this.transactionService.submitSignedTransaction(
        dto.serializedTransaction,
        dto.transactionId,
    );
    return { data: result };
}
```

### 2.7.2 Impact

A user can submit any transaction they signed, as long as they pick a pending DB transactionId whose from matches their wallet > DB/intent integrity issues (and potential business-logic bypass depending on how on-chain events map back to DB records).

### 2.7.3 Recommendation

- Store and enforce an expected transaction digest (or canonical hash) at build time.
- Parse and validate the submitted transaction against the DB record (program ID, instruction type, accounts, amounts, fee payer).

## 2.8 Refresh Tokens Can Be Used As Bearer Access Tokens (Token-Type Confusion)

JwtStrategy treats tokens containing refreshTokenJti as valid auth for AuthGuard("jwt"). This allows long-lived refresh tokens to act like access tokens.

### 2.8.1 Code location

- File Location: jwt.strategy.ts

```
  async validate(payload: JwtPayload): Promise<JwtPayload> {
      const { id, refreshTokenJti } = payload;

      let user: User;

      if (refreshTokenJti) {
          user = await User.findOne({ where: { refreshTokenJti } });
          if (!user) throw new UnauthorizedException("User not found by refresh token");
      } else {
          user = await User.findOne({ where: { id } });
          if (!user) throw new UnauthorizedException("User not found by id");
      }

      if (user.status === UserStatus.BLOCKED)
          throw new ForbiddenException("user is blocked. please contact support");

      return { id, refreshTokenJti };
  }
```

### 2.8.2 Impact

If a refresh token leaks (even once), the attacker can call protected endpoints directly until refresh expiry (bypassing intended “short access token” security).

### 2.8.3 Recommendation

- Add a required claim (e.g. typ: "access") and reject non-access tokens in JwtStrategy.
- Or use separate secrets for access vs refresh and different passport strategies.

## 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

- File Location: cli.ts lines 5-20

```
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 Logout Does Not Revoke Refresh Tokens Server-Side Due To Refresh Cookie Path Scoping

The refresh token is stored in an HttpOnly cookie set with Path=/auth/refresh-token. Browsers only send cookies to requests whose path matches the cookie Path, so POST /auth/logout typically will not receive refreshToken. Since the logout handler only revokes the refresh token in DB if req.cookies.refreshToken is present, logout often becomes client-only (cookie cleared) and does not revoke server-side refresh validity (refreshTokenJti stays valid).

### 2.10.1 Code location

- File Location: Auth.controller.ts lines 45-57

```
private setRefreshTokenCookie(res: Response, refreshToken: string): void {
    const cookieMaxAge = process.env.REFRESH_TOKEN_LIFESPAN_MIN
        ? Number(process.env.REFRESH_TOKEN_LIFESPAN_MIN) * 60 * 1000
        : 1 * 24 * 60 * 60 * 1000; // Default: 1 day in milliseconds

    res.cookie("refreshToken", refreshToken, {
        httpOnly: true,
        sameSite: "lax",
        secure: process.env.ENVIRONMENT === "production",
        maxAge: cookieMaxAge,
        path: "/auth/refresh-token",
    });
}
```

### 2.10.2 Impact

- Server-side session persistence: “Logout” may not invalidate refresh capability; the account remains refreshable server-side until expiry/rotation.
- Stolen refresh token remains usable after logout: if an attacker has a copy of the refresh token, they can keep refreshing even after the victim “logs out”.
- False security guarantee: users/admins may believe logout terminates sessions when it often does not.

### 2.10.3 Recommendation

To mitigate this issue, it is recommended to implement one of the following improvements:

- Broaden cookie path: Set refresh cookie Path=/auth (or /) so it is sent to /auth/logout, enabling DB revocation. Keep using the same Path when clearing it.
- Align logout route to the cookie Path: e.g. add POST /auth/refresh-token/logout (or move logout under that path) so the browser will send the refresh cookie without broadening cookie exposure.
- Revoke by authenticated user: protect logout with AuthGuard("jwt") and revoke by req.user.id (set refreshTokenJti = null) regardless of whether the cookie is present, while still clearing the cookie client-side.

As an additional hardening step, it is recommended to verify the refresh token cryptographically during logout (avoid relying on decode()), so revocation cannot be triggered from untrusted, non-verified token payloads.

## 2.11 Global Validation Does Not Whitelist/Strip Unknown Fields (Mass-Assignment Risk)

ValidationPipe is global, but no whitelist, forbidNonWhitelisted, or transform is configured. This commonly enables “extra-field injection” into DTOs, which can become privilege escalation if any service merges DTOs into entities.

### 2.11.1 Code location

- File Location: main.ts

```
const validationConfig = new CustomValidationPipe();

app.enableCors({
    origin: process.env.CORS_ORIGINS ? process.env.CORS_ORIGINS.split(",") : "*",
    methods: ["GET", "PUT", "POST", "DELETE", "PATCH"],
    credentials: true,
});

app.useGlobalPipes(new ValidationPipe(validationConfig));
```

- File Location: custom-validation-options.ts

```
export class CustomValidationPipe implements ValidationPipeOptions {
    public readonly disableErrorMessages: boolean = false;
    public readonly errorHttpStatusCode: number = 400;
    public exceptionFactory? = (errors: ValidationError[]): BadRequestException => {
        const errorList = {};
        errors.forEach((error) => {
            const constraints = Object.keys(error.constraints)
                .map((key) => error.constraints[key].replace(",", ""))
                .join(", ");

            errorList[`${error.property}`] = constraints;
        });
        return new BadRequestException({
            errors: errorList,
```

### 2.11.2 Impact

Depending on downstream DTO > entity handling, attackers may be able to inject/override fields that should be server-controlled (status/roles/IDs/wallets/etc.).

### 2.11.3 Recommendation

To mitigate this issue, it is recommended to implement following improvements:

- Enable whitelist: true and ideally forbidNonWhitelisted: true.
- Enable safe transform: true if you rely on typed DTO coercion.
- Audit all DTO > entity writes for Object.assign, spread merges, or passing raw bodies to ORM .save().

## 2.12 CORS Defaults To * With Credentials: True (Dangerous Default)

If CORS_ORIGINS isn’t set, the API enables CORS for all origins while allowing credentials.

### 2.12.1 Code location

- File Location: main.ts

```
app.enableCors({
   origin: process.env.CORS_ORIGINS ? process.env.CORS_ORIGINS.split(",") : "*",
   methods: ["GET", "PUT", "POST", "DELETE", "PATCH"],
   credentials: true,
```

### 2.12.2 Impact

In browser contexts, this expands cross-origin attack surface and can lead to credential leakage/misbehavior depending on deployment and proxies.

### 2.12.3 Recommendation

To mitigate this issue, it is recommended to implement following improvements:

- Require an explicit allowlist in all non-local environments.
- Prefer an origin validation callback that rejects unknown origins.

## 2.13 Floating-Point Math Used For U64 Token Amounts/ Multipliers (Precision/Overflow + Rounding Ambiguity)

Critical values passed on-chain are derived from JS floating point numbers (number), e.g. Math.floor(amount * USDC_SCALE) and new BN(cashOutMultiplier * SC_BET_CRASH_SCALE). Float precision loss or non-integer inputs can create incorrect u64s, leading to unexpected on-chain amounts/multipliers and reconciliation issues.

### 2.13.1 Code location

- File Location: solana-transaction-builder.service.ts lines 188-199

```
const userPubkey = new PublicKey(walletAddress);
const amountLamports = Math.floor(amount * USDC_SCALE);
this.logger.log(`Building DEPOSIT transaction: ${amount} USDC for ${walletAddress}`);
const transaction = await this.anchorProgram.methods
    .deposit(new BN(amountLamports))
    .accounts({
        usdcMint: this.usdcMint,
        authority: userPubkey,
    })
    .transaction();
```

### 2.13.2 Impact

Amount/multiplier mismatches, payout/bet precision drift, potential edge-case abuse, and brittle reconciliation between backend expectations and on-chain truth.

### 2.13.3 Recommendation

To mitigate this issue, it is recommended to:

- Treat amounts as decimal strings and parse to exact integer units (e.g., BigInt or a decimal library), enforcing max 6 decimals for USDC.
- Validate all numeric inputs: finite, positive, within u64 bounds, and integer at the unit level before constructing BN.
- Standardize rounding rules across bet/deposit/withdraw/payout.

## 2.14 Unbounded Limit On Network List Queries

NetworkFilterDto.limit has @Min(1) but no max, and the service passes it to TypeORM take via getPagination(...) without clamping.

### 2.14.1 Code location

- File Location: network-response.dto.ts lines 9-35

```
export class NetworkFilterDto {
   // ...
   @ApiPropertyOptional({ description: "Items per page", default: 20 })
   @IsOptional()
   @Type(() => Number)
   @IsNumber()
   @Min(1)
   limit?: number;
```

### 2.14.2 Impact

Large limit > heavy DB queries/large responses > performance degradation.

### 2.14.3 Recommendation

To mitigate this issue, it is recommended to add @Max(<reasonable>) (e.g., 100) and also clamp server-side (limit = Math.min(limit, 100)).

# 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 Supply-chain & secret handling (.npmrc, Docker build)

**What is checked**

- No GitHub Packages token is committed to source (.npmrc uses env token only).
- Docker images do not contain .npmrc or _authToken remnants.
- Installs use BuildKit secrets (token never appears in image layers).

**Scenarios**

- **TC-SUPPLY-01 - Repo secret scan**
  - Search for hardcoded tokens (e.g., ghp_, _authToken=").
  - Confirm .npmrc contains only ${NODE_AUTH_TOKEN}-style interpolation.
- **TC-SUPPLY-02 - Image artifact inspection**
  - Build image with BuildKit secrets enabled.
  - Inspect final image filesystem and layers for .npmrc / _authToken.
- **TC-SUPPLY-03 - “Fail safe” build behavior**
  - Attempt a build without providing the npm_token secret.

**Secure expected outcome**

- No committed token exists; final runtime image contains no .npmrc and no token string.
- Builds without secrets fail (or cannot install private packages), rather than “fallback to embedded creds”.

**Evidence to capture**

- Secret scan output (paths + matches/no-matches).
- Image grep output + docker history excerpts showing no token materialization

## 3.2 Dev / debug exposure (dev endpoints + guards)

**What is checked**

- Dev-only endpoints are not reachable in production deployments.
- Dev-only access control fails closed under misconfiguration.

**Scenarios**

- **TC-DEV-01 - Prod: /dev/* not registered**
  - Run with ENVIRONMENT=production.
  - Request GET /dev/mint and GET /dev/generate-token.
- **TC-DEV-02 - Missing env fails closed**
  - Start the app with ENVIRONMENT unset (or invalid).
- **TC-DEV-03 - Dev/test: endpoints exist only when intended**
  - Run with ENVIRONMENT=development or test.
  - Confirm /dev/* exists (for local testing only) and does not silently appear in prod mode.

**Secure expected outcome**

- Production: /dev/* returns 404 (module not loaded), not “accidentally enabled”.
- Missing/invalid env causes startup failure (no implicit default to development/test).

**Evidence to capture**

- HTTP responses (status + route behavior) from prod-like config.
- Startup logs showing env validation failures when misconfigured.

## 3.3 Listener pipeline integrity (failed on-chain tx handling)

**What is checked**

- Failed Solana transactions (meta.err) do not emit reconciliation events and cannot drive DB state changes.

**Scenarios**

- **TC-SOL-01 - Forced failed tx**
  - Submit a transaction that deterministically fails on-chain (e.g., insufficient funds / invalid accounts).
  - Feed its signature into the normal sync/parser path (or wait for sync to ingest it).
  - Verify DB does not apply “success-path” effects for that signature.
- **TC-SOL-02 - Historical replay safety**
  - Re-run historical scanning over a range containing failed txs; ensure they remain non-impacting.

**Secure expected outcome**

- No success events are emitted/handled for signatures with meta.err.
- DB remains consistent with on-chain truth.

**Evidence to capture**

- Solana explorer view showing meta.err.
- Application logs showing “failed on-chain” handling.
- DB diff/queries showing no credited deposit/withdraw/bet effects from that signature.

## 3.4 Transaction intent binding (POST /transactions/submit-user-signed)

**What is checked**

- The submitted user-signed transaction is bound to backend intent (cannot swap instructions/accounts/amounts under the same transactionId).
- Replay and cross-transaction substitution fails safely.

**Scenarios**

- **TC-TX-01 - Swap content under same transactionId**
  - Build a tx via POST /transactions/build/* to get { serializedTransaction, transactionId }.
  - Decode, modify a critical field (amount / destination / instruction), re-sign, submit via POST /transactions/submit-user-signed.
- **TC-TX-02 - Submit tx A using txId B**
  - Build two different transactions; submit serialized tx A while referencing transactionId B.
- **TC-TX-03 — ComputeBudget-only change**
  - Modify only ComputeBudget instructions (priority fee / CU limit), sign, submit.

**Secure expected outcome**

- Any substantive mutation (accounts/amounts/instruction set) is rejected with a stable error (e.g., message hash mismatch) and does not get submitted.
- ComputeBudget-only changes are tolerated (if intentionally excluded from the message hash), while core intent remains protected.

**Evidence to capture**

- Request/response bodies and status codes.
- Logs showing mismatch detection and any DB state update to FAILED.

## 3.5 Authentication (SIWS + JWT) and session security

**What is checked**

- No JWT “default secret” fallback exists; secrets are required.
- Access tokens and refresh tokens are cryptographically separated and type-enforced.

**Scenarios**

- **TC-AUTH-01 - Startup without secrets**
  - Run without JWT_ACCESS_SECRET / JWT_REFRESH_SECRET.
- **TC-AUTH-02 - Refresh token used as access token**
  - Obtain a refresh token (test client / captured cookie).
  - Present it as Authorization: Bearer <refreshToken> to a JWT-protected endpoint (e.g., GET /users/me).
- **TC-AUTH-03 - Tampered typ**
  - Modify token payload typ and verify rejection.

**Secure expected outcome**

- App refuses to start without secure secrets.
- Refresh tokens cannot authenticate to access-protected routes (wrong secret and/or wrong typ).

**Evidence to capture**

- Startup failure logs for missing secrets.
- 401/403 responses for wrong token types.

## 3.6 Refresh + logout semantics

**What is checked**

- Refresh token rotation is reuse-resistant.
- Logout can revoke the caller’s refresh capability (cookie path scoping is correct).

**Scenarios**

- **TC-SESS-01 - Rotation / reuse**
  - Call POST /auth/refresh-token and capture the old refresh token value.
- **Attempt refresh again using the old token.**
- **TC-SESS-02 - Logout revocation**
  - With refresh cookie present, call POST /auth/logout.
  - Attempt POST /auth/refresh-token using the same refresh token after logout.

**Secure expected outcome**

- Old refresh tokens fail after rotation.
- Post-logout refresh fails (server-side revocation effective).
- Cookie path allows /auth/logout to receive refresh cookie (Path=/auth behavior).

**Evidence to capture**

- Cookie attributes in responses.
- Refresh success then old-token failure.
- Logout then refresh failure.

## 3.7 Referral commission integrity

**What is checked**

- Commission dedup is stable (no depth mismatch duplicates).
- Missing referral owner data fails closed (no "" address pollution).

**Scenarios**

- **TC-REF-01 - Duplicate processing**
  - Reprocess the same payout/commission event (or same signature) twice.
  - Compare referral_commission row counts and values before/after.
- **TC-REF-02 - Missing owner wallet**
  - Induce referral owner lookup failure (e.g., RPC failure injection / invalid referral PDA in a controlled test).
  - Ensure parsing fails and no “empty wallet user” is created.

**Secure expected outcome**

- Reprocessing updates/merges existing commission rows instead of inserting duplicates.
- No commission/user records are created for invalid/empty referral owners.

**Evidence to capture**

- DB queries showing stable uniqueness by (betId, ancestorId, descendantId, depth) semantics.
- Logs showing parse failure for missing ownerWallet (and no follow-on user creation).

## 3.8 Input validation, parsing robustness, and ambiguity handling

**What is checked**

- Unknown fields are rejected at the boundary and cannot reach business logic.

**Scenarios**

- **TC-VAL-01 - Unknown fields**
  - Send extra properties in JSON bodies (e.g., role, status, userId, etc.) to representative endpoints.
- **TC-VAL-02 - Type confusion**
  - Send strings/objects for numeric fields and observe validation behavior.

**Secure expected outcome**

- Requests with unknown fields are rejected with 4xx (forbid-non-whitelisted).
- No silent field stripping that could mask abuse.

**Evidence to capture**

- Validation error responses showing unknown fields are rejected.

## 3.9 Web security posture (CORS)

**What is checked**

- CORS origins are explicitly controlled per environment (no implicit permissive fallback).

**Scenarios**

- **TC-CORS-01 - Untrusted origin**
  - Send requests with Origin: https://attacker.example to credentialed endpoints.
- **TC-CORS-02 - Allowed origin**
  - Repeat from an origin included in CORS_ORIGINS.

**Secure expected outcome**

- Untrusted origins do not receive permissive CORS headers.
- Allowed origins behave as expected for credentialed flows.

**Evidence to capture**

- Response headers: Access-Control-Allow-Origin, Access-Control-Allow-Credentials.

## 3.10 Availability / DoS resistance (paging bounds)

**What is checked**

- Expensive list endpoints enforce hard bounds (no unbounded limit).

**Scenarios**

- **TC-DOS-01 - Oversized limit**
  - Call GET /networks/level/:level/... with limit=100000.
- **TC-DOS-02 - Boundary behavior**
  - Verify limit=100 succeeds and limit=101 fails.

**Secure expected outcome**

- Oversized limit is rejected at validation time with 4xx.
- DB queries do not execute with attacker-chosen huge take.

**Evidence to capture**

- 4xx responses for oversize limits + logs showing no heavy query executed.

## 3.11 CLI tooling exposure (dev-only commands)

**What is checked**

- High-privilege dev CLI commands are not available in production environments.

**Scenarios**

- **TC-CLI-01 - Prod env: dev commands absent**
  - Run CLI with ENVIRONMENT=production and attempt to invoke dev commands (debug tx / send tx / check tx build).
- **TC-CLI-02 - Dev env: commands available only for local testing**
  - Run CLI with ENVIRONMENT=development|test and verify dev commands exist (staging-local only).

**Secure expected outcome**

- Production env does not expose dev command entry points (or they are inert).
- Dev commands are strictly limited to non-production environments.

**Evidence to capture**

- CLI help output in prod (no dev commands listed).
- CLI help output in dev/test (commands listed).

IRONNODE

ironnode.io

# Thank you for choosing us!