Vulnerability Assessment Report

rings

November 17, 2026 — Whitebox Scan

2	2	1
HIGH	MEDIUM	LOW

Table of Contents

Executive Summary 3

Recommendations 4

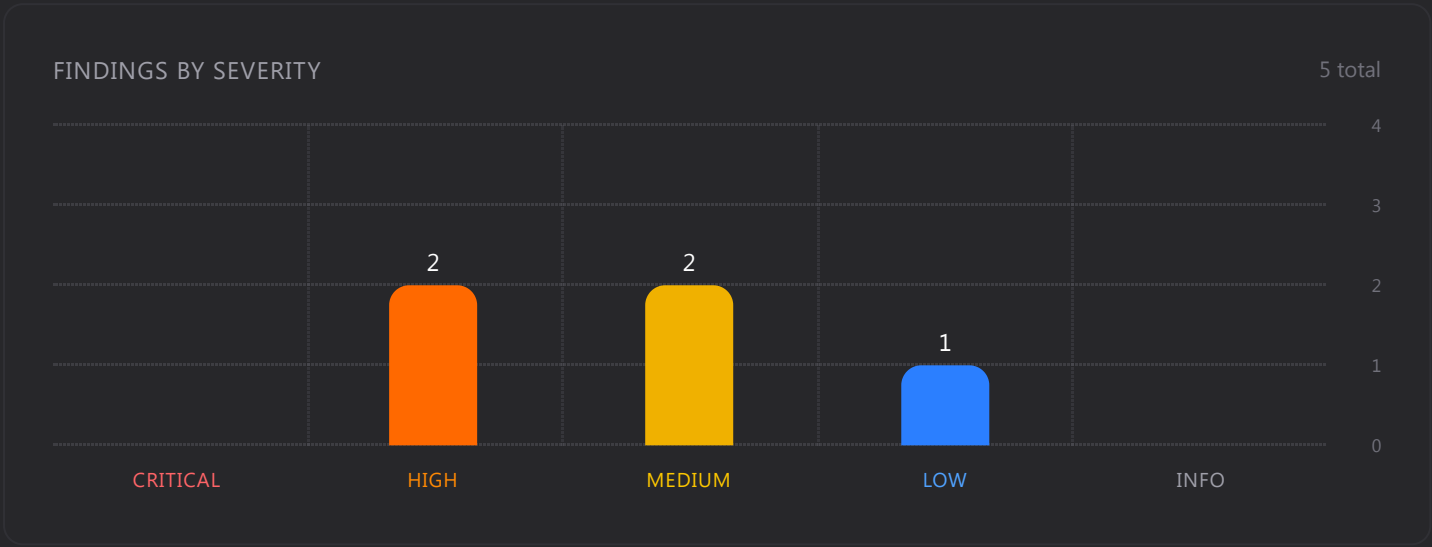
Detailed Findings 6

Other 6

SECTION 01

Executive Summary

This assessment of rings identified 5 findings across 1 category. No critical-severity issues were found.



SEVERITY DEFINITIONS

- CRITICAL — Trivially or remotely exploitable with severe impact — data breach, account or fund takeover, remote code execution. Fix immediately.
- HIGH — A serious, exploitable weakness with significant impact that a motivated attacker would likely abuse. Fix urgently.
- MEDIUM — A real weakness with limited impact or one that needs specific conditions or authentication to exploit. Fix soon.
- LOW — A minor issue or missing hardening with little direct impact on its own. Address opportunistically.
- INFO — An informational observation or best-practice note with no direct security impact.

METHODOLOGY

Source-driven (white-box) assessment with full read access to the application's code. The scanner clones the repository at `main` and builds a static call-graph with taint analysis to trace untrusted input into security-sensitive sinks (queries, auth checks, access-control rules, external calls). An LLM triages each reachable path to separate real vulnerabilities from noise; backend-specific runtime probes then confirm configuration and access-control issues against the live deployment, and exploitable paths are proven end-to-end with real requests. Every confirmed finding is de-duplicated to a single root cause and scored with CVSS. This examines logic and data-flow a black-box scan cannot see from the outside — the reason white-box surfaces materially more, and more deeply proven, issues.

FINAL SECTION

Recommendations

Prioritized by severity — address critical and high items first.

HIGH

Unbounded rate-limit Map → process OOM crash (crash-loop outage)

Immediate fix: bound the limiter. Cap entries and sweep expired on a timer, or move to a fixed-window counter in Redis so memory is O(active keys), not O(all keys ever seen).

```
// src/server/http.ts — bounded + swept
const hits = new Map<string, { count: number; resetAt: number }>();
const MAX_KEYS = 100_000;
setInterval(() => {
  const now = Date.now();
  for (const [k, h] of hits) if (h.resetAt < now) hits.delete(k);
}, 30_000).unref();
export function rateLimit(key: string, max: number, windowMs: number) {
  const now = Date.now();
  const h = hits.get(key);
  if (!h || h.resetAt < now) {
    if (hits.size >= MAX_KEYS) hits.clear(); // hard ceiling
    return void hits.set(key, { count: 1, resetAt: now + windowMs });
  }
  if (++h.count > max) throw new ApiError(429, "Too many requests. Wait a moment.");
}
```

Fixing Finding 1 (trusted key) also removes the unbounded-key-space precondition.

HIGH

Unauthenticated correlated-query list endpoints saturate a max:4 DB pool

Immediate fix: short-TTL cache the unauthenticated list endpoints (the public open-blocks list changes on the order of seconds, not per-request), and add a global concurrency cap so list traffic cannot consume every pool connection.

```
// cache /api/blocks for a few seconds across all callers
let cached: { at: number; body: string } | null = null;
const TTL = 3_000;
if (cached && Date.now() - cached.at < TTL) return new Response(cached.body);
// ... run query, then: cached = { at: Date.now(), body };
```

Also: raise/segment the pool (`max`) for read traffic, confirm the production DB engine (PGlite vs managed Postgres), and paginate the `limit 200` list.

MEDIUM

X-Forwarded-For spoofing defeats all per-IP rate limiting

Immediate fix: stop trusting the client-supplied first XFF token. Key rate limits on the trusted peer identity your edge sets (Cloudflare: `CF-Connecting-IP`), and configure the edge to STRIP any client-supplied `X-Forwarded-For` before it reaches the origin.

```
// src/server/http.ts
export const clientIp = (req: Request) =>
  req.headers.get("cf-connecting-ip") ||           // trusted, edge-set
  req.headers.get("x-real-ip") ||
  "local";                                         // never trust raw x-forwarded-for[0]
```

Defense in depth: fleet origin-lock (reject requests not carrying the edge shared-secret header) so attackers cannot hit the origin directly and forge XFF at all.

MEDIUM

Helius RPC proxy: unauthenticated paid-key quota/cost drain

Immediate fix: put a GLOBAL (not per-IP) budget on the proxies and gate them behind an authenticated session; specifically rate-limit the registry scan, which is the expensive allowlisted call.

```
// src/app/api/helius/rpc/route.ts
rateLimit(`rpc:${clientIp(req)}`, 900, 60_000);     // per-IP (bypassable)
rateLimit("rpc:global", 6_000, 60_000);            // hard global ceiling on paid-key spend
if (isRegistryScan(call)) rateLimit("rpc:registry-global", 120, 60_000);
```

LOW

DEX swap/order quoting self-throttles on shared free Jupiter tier

Use a Jupiter API key (dedicated quota), add a brief server-side quote cache keyed on (inputMint,outputMint,amount,slippage), and add a per-wallet quote budget so one user cannot exhaust the shared tier.

SECTION 03

Other 5 findings

HIGH

Unbounded rate-limit Map → process OOM crash (crash-loop outage)

`any rate-limited route (keyed by clientIp)`

95% Confidence

7.5 CVSS

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

Tier: Critical (real crash reproduced - FATAL heap OOM, exit 134).

The rate limiter stores one entry per key in a module-level `Map` that is NEVER swept - expired entries are only overwritten on a same-key re-access. Chained with Finding 1 (attacker controls the key via rotating XFF), every request mints a permanent entry. Linear growth (~180 B/entry) exhausts the heap; under a process manager this is a crash-loop = sustained outage. Unauthenticated, one machine.

EVIDENCE

Isolated verbatim-code harness: 180 B/entry, linear (1M->176MB ... 4M->692MB); 200k entries @1ms window +50ms wait -> Map.size unchanged (never swept). Under node --max-old-space-size=200: "FATAL ERROR: Reached heap limit Allocation failed - JavaScript heap out of memory", exit 134.

REPRODUCTION

```
# standalone, verbatim hits+rateLimit from http.ts
node --max-old-space-size=200 poc/poc_A_oom.mjs
# => feeds unique IPs -> FATAL ERROR: Reached heap limit ... exit 134
```

Growth-only (no crash): node --expose-gc poc/poc2_leak.mjs

HIGH Unbounded rate-limit Map → process OOM crash (crash-loop outage) *continued*

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

REMEDIATION

Immediate fix: bound the limiter. Cap entries and sweep expired on a timer, or move to a fixed-window counter in Redis so memory is O(active keys), not O(all keys ever seen).

```
// src/server/http.ts — bounded + swept
const hits = new Map<string, { count: number; resetAt: number }>();
const MAX_KEYS = 100_000;
setInterval(() => {
  const now = Date.now();
  for (const [k, h] of hits) if (h.resetAt < now) hits.delete(k);
}, 30_000).unref();
export function rateLimit(key: string, max: number, windowMs: number) {
  const now = Date.now();
  const h = hits.get(key);
  if (!h || h.resetAt < now) {
    if (hits.size >= MAX_KEYS) hits.clear(); // hard ceiling
    return void hits.set(key, { count: 1, resetAt: now + windowMs });
  }
  if (++h.count > max) throw new ApiError(429, "Too many requests. Wait a moment.");
}
```

Fixing Finding 1 (trusted key) also removes the unbounded-key-space precondition.

HIGH

Unauthenticated correlated-query list endpoints saturate a max:4 DB pool

`GET /api/blocks, GET /api/requests`

90% Confidence

7.5 CVSS

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

Tier: Exploited (grounded on real Postgres; latency collapse measured).

`GET /api/blocks` (multiple correlated subqueries) and `/api/requests` run unauthenticated, uncached, behind a `pg` pool capped at `max: 4`. Query cost scales with table size, so a modest number of open blocks plus ~150 concurrent requests exhausts the 4 connections and collapses the whole app. Chained with Finding 1, one attacker sustains it. Engine-independent (a single-threaded PGlite prod DB would be worse).

EVIDENCE

Real Postgres: empty DB -> 7ms single / 319ms p95@150conc / ~459 req/s. 2000 open blocks -> 1005ms single / 37965ms p95@150conc / ~7 req/s ceiling. One unauth request ~1s; 150 concurrent -> 38s responses, whole app at 7 req/s.

REPRODUCTION

```
DBURL=<app DATABASE_URL> N=2000 node poc/seedB.mjs   # seed open blocks
node poc/measB.mjs                                    # single + conc 1/20/60/150 latency
# cleanup: DELETE FROM blocks WHERE seller='POCSEEDSELLER';
```

REMEDIATION

Immediate fix: short-TTL cache the unauthenticated list endpoints (the public open-blocks list changes on the order of seconds, not per-request), and add a global concurrency cap so list traffic cannot consume every pool connection.

```
// cache /api/blocks for a few seconds across all callers
let cached: { at: number; body: string } | null = null;
const TTL = 3_000;
if (cached && Date.now() - cached.at < TTL) return new Response(cached.body);
// ... run query, then: cached = { at: Date.now(), body };
```

Also: raise/segment the pool (`max`) for read traffic, confirm the production DB engine (PGlite vs managed Postgres), and paginate the `limit 200` list.

MEDIUM

X-Forwarded-For spoofing defeats all per-IP rate limiting

`POST /api/auth/nonce (and every rate-limited route)`

97% Confidence

5.3 CVSS

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L

Tier: Exploited (runtime-confirmed - live bypass fired on the local stack).

`clientIp()` derives the rate-limit key from the FIRST token of the client-supplied `X-Forwarded-For` header. An attacker rotates that token per request to mint a fresh limiter bucket every time, so no per-IP limit ever trips. Behind Cloudflare the first XFF token stays attacker-controlled (CF *appends* to inbound XFF), so this is not closed by the CDN unless the edge strips client XFF or the app keys on CF-Connecting-IP. This is the ROOT finding: it unlocks the OOM crash (A) and the DB-pool collapse (B).

EVIDENCE

POST /api/auth/nonce, limit 30/60s/IP: fixed XFF -> 200x30 then 429x30; rotating XFF -> 200x60, 429x0 (limit never trips).

REPRODUCTION

```
# fixed XFF trips the limiter; rotating XFF never does
for i in $(seq 1 60); do
  curl -s -o /dev/null -w "%{http_code}\n" -X POST http://localhost:3100/api/auth/nonce \
    -H "X-Forwarded-For: $((RANDOM%255)).$((RANDOM%255)).$((RANDOM%255)).$((RANDOM%255))"
done | sort | uniq -c   # => 60x 200, 0x 429
```

See also poc/poc.mjs (Finding 1).

REMEDIATION

Immediate fix: stop trusting the client-supplied first XFF token. Key rate limits on the trusted peer identity your edge sets (Cloudflare: `CF-Connecting-IP`), and configure the edge to STRIP any client-supplied `X-Forwarded-For` before it reaches the origin.

```
// src/server/http.ts
export const clientIp = (req: Request) =>
  req.headers.get("cf-connecting-ip") ||           // trusted, edge-set
  req.headers.get("x-real-ip") ||
  "local";                                         // never trust raw x-forwarded-for[0]
```

Defense in depth: fleet origin-lock (reject requests not carrying the edge shared-secret header) so attackers cannot hit the origin directly and forge XFF at all.

MEDIUM

Helius RPC proxy: unauthenticated paid-key quota/cost drain

`POST /api/helius/rpc, POST /api/helius/spr`

80% Confidence

5.3 CVSS

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L

Tier: Confirmed (precondition proven; not flooded to real quota exhaustion).

The Helius proxies forward with the server's PAID key and are gated only per-IP (900/min, 600/min) with no app auth - so Finding 1 removes the only limit. The one allowlisted `getProgramAccounts` (the SPP registry scan) is an expensive full-program scan an attacker can repeat to burn Helius quota / rack cost.

EVIDENCE

30 rotate-XFF POSTs to /api/helius/rpc -> 401x30 (reached upstream past the limiter; no app auth; same clientIp key as Finding 1). Not flooded to 900+/min / did not drain real quota.

REPRODUCTION

```
# reaches the paid upstream with no app-level auth, limiter bypassed via rotating XFF
for i in $(seq 1 30); do
  curl -s -o /dev/null -w "%{http_code}\n" -X POST http://localhost:3100/api/helius/rpc \
    -H "X-Forwarded-For: $((RANDOM%255)).$((RANDOM%255)).$((RANDOM%255)).$((RANDOM%255))" \
    -H "content-type: application/json" -d '{"method":"getHealth","params":[]}'
done
```

REMEDIATION

Immediate fix: put a GLOBAL (not per-IP) budget on the proxies and gate them behind an authenticated session; specifically rate-limit the registry scan, which is the expensive allowlisted call.

```
// src/app/api/helius/rpc/route.ts
rateLimit(`rpc:${clientIp(req)}`, 900, 60_000);     // per-IP (bypassable)
rateLimit("rpc:global", 6_000, 60_000);            // hard global ceiling on paid-key spend
if (isRegistryScan(call)) rateLimit("rpc:registry-global", 120, 60_000);
```



LOW

DEX swap/order quoting self-throttles on shared free Jupiter tier

`POST /api/dex/swaps, POST /api/dex/orders`

60% Confidence

3.7 CVSS

CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:L

Tier: Detected (code-only - not run; DEX disabled on the audited boot).

Each DEX swap/order does a live Jupiter quote against the shared free endpoint (`lite-api.jup.ag` , no API key). A burst earns Jupiter-side 429s, so quotes fail for all users (graceful back-off, not a crash). Lower severity: self-inflicted degradation, not attacker-controlled outage.

EVIDENCE

Static: swapper.ts quotes via https://lite-api.jup.ag with no API key; per-request live quote on the shared free tier. Not exercised (DEX off on the audited standup).

REPRODUCTION

Code-only (not run). A burst of POST /api/dex/swaps would drive the shared free Jupiter tier to 429, failing quotes app-wide.

REMEDIATION

Use a Jupiter API key (dedicated quota), add a brief server-side quote cache keyed on (inputMint,outputMint,amount,slippage), and add a per-wallet quote budget so one user cannot exhaust the shared tier.