ZEALYNX

Web3 Security and AI Agent & MCP Audits

Dripster

Backend Pentesting

29 April 2026

Zealynx
contact@zealynx.io

Fernando @0xMrjory

Contents

1. About Zealynx

Zealynx, founded in January 2024 by Carlos (Bloqarl), specializes in smart contract audits, and development. Our services include comprehensive smart contract audits, application security audits, such as pentesting, and AI Audits. We are trusted by clients such as Badger DAO, Ample protocol, Lido, Inverter, Matchain, and Golden Grid.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website Zealynx.io and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

This penetration testing assessment was conducted within the defined scope and timeframe agreed upon with the client. While every effort was made to identify security vulnerabilities, this assessment cannot guarantee the discovery of all weaknesses or ensure complete security of the tested systems. The findings represent a point-in-time evaluation; subsequent changes to systems, configurations, or the threat landscape may introduce new vulnerabilities. The assessor assumes no liability for security incidents occurring after the assessment or for vulnerabilities outside the agreed scope. This report is confidential and intended solely for the authorized recipient. Testing was performed with proper authorization and in compliance with applicable laws and regulations.

3. Overview

3.1 Project Summary

A penetration testing assessment of the Dripster backend (the NestJS / Prisma API at `bloom-art/api`) was conducted by Zealynx Security with focus on authentication, authorization, business logic, input validation, and infrastructure hygiene.

The engagement spanned 10 days against a baseline commit locked at the start of the window, with findings filed in real time as GitHub issues so the team could begin remediation before delivery. The assessment surfaced 21 issues, comprising 1 High, 4 Medium, 11 Low, and 5 Informational findings.

3.2 About Dripster

Dripster (Dimes.fi) is a leveraged prediction-market vault on Polygon. The backend coordinates the off-chain order routing on Polymarket's CLOB, manages the position lifecycle, and signs the EIP-712 payloads that gate value-changing operations on the on-chain `LeveragedPredictionVaultV1` contract.

3.3 Audit Scope

The assessment covered the Dripster backend at:

- Repository: `github.com/bloom-art/api`
- Branches: `dev`
- Engagement window: 13 April 2026 – 23 April 2026
- Surfaces in scope: customer / partner / admin REST controllers, authenticated WebSocket gateways, webhook receivers (QuickNode, Alchemy, Telegram), oracle and lending control planes, repository configuration (Docker Compose, env files, gitleaks)

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain **high-quality standards**. Our methodology encompasses thorough **Backend Code Review**, **API Security Testing**, **Infrastructure Assessment**, and meticulous manual **penetration testing**.

Throughout the TypeScript application audit and penetration testing process, we prioritize the following aspects to uphold excellence:

1. **Backend Architecture Analysis:** Our assessments emphasize secure coding practices, authentication mechanisms, authorization controls, session management, and data validation to ensure robust server-side security.
2. **API Security Testing:** We probe each authenticated and unauthenticated endpoint against the OWASP API Security Top 10 (2023), covering injection vectors, broken authorization, parameter tampering, mass assignment, and unsafe consumption of upstream APIs.
3. **Infrastructure and Configuration:** We meticulously review deployment configurations, environment variables, database connections, secret management, and third-party integrations to identify misconfigurations and security gaps.
4. **Penetration Testing:** We conduct both automated and manual testing against OWASP Top 10 vulnerabilities and custom DeFi / prediction-market threat scenarios, including real-world exploitation attempts to demonstrate actual risk impact.

Testing Execution

During the Dripster engagement, Zealynx performed the following actions to ensure thorough coverage of the six SoW-defined testing areas (authentication, business logic, input validation, rate limiting, infrastructure, and Polymarket integration):

- **Authentication.** Mapped the authentication surface (customer JWT via Privy, partner API keys, internal admin / partnerAdmin keys) and exercised the resource-ownership guard chain across customer / partner / admin / oracle / lending controller groups.

Probed for role escalation, JWT-validation gaps, API-key management flaws, and session-handling weaknesses.

- **Business Logic.** Performed walkthroughs of the offer / quote / position lifecycle, the circuit-breaker pause tiers, leverage-boundary enforcement, and the settlement flow — including the `bypassChecks` branch, the cache-clear admin surface, and the Polymarket integration boundary.
- **Input Validation.** Probed the customer / partner / admin endpoints for SQL injection vectors (focused on Prisma raw-SQL surfaces), parameter tampering on URL path params and request bodies, type confusion against Zod / class-validator schemas, and mass-assignment via `CamelizeBodyPipe`.
- **Rate Limiting.** Reviewed the `@Throttle`-annotated endpoints and the throttler guard's identity-distinction logic across the JWT / API-key / IP fallback chain.
- **Infrastructure.** Reviewed Docker Compose configuration, environment file management, committed secrets, the gitleaks allowlist posture, CORS posture, and webhook signature implementations (QuickNode, Alchemy, Telegram) for replay protection, timestamp validation, and HMAC integrity.
- **Polymarket Integration.** Reviewed the external API interaction surface — Polymarket builder credential storage, the V2 builder-code transition, and the data-integrity invariants between the off-chain CLOB order routing and the on-chain finalization path.
- **Manual code review.** Conducted line-by-line review of the controller, guard, service, and repository layers, focusing on authorization boundaries, signature verification, encryption primitives, and Prisma raw-SQL surfaces.

Validation Process

After initial testing and reporting, Zealynx followed a structured process to ensure the accuracy and effectiveness of remediation:

- Documented all findings with supporting evidence, impact analysis, CVSS 3.1 scores, and actionable remediation guidance.
- Filed findings as GitHub issues in real time so the team could engage with each item before final delivery.

- For each issue marked as resolved by the team, re-tested the affected components against the current HEAD to confirm successful mitigation and closure.
- Performed a baseline-versus-final-state delta review on 21–23 April: each finding's Status section in Section 7 documents both the original (baseline) behavior and the final state at delivery, distinguishing between issues confirmed as fixed and those acknowledged with rationale.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Over a 10-day engagement, the Zealynx Security team conducted a penetration testing assessment of the Dripster backend (the NestJS / Prisma API at `bloom-art/api`). The assessment was conducted from 13 to 23 April 2026 against a baseline commit locked at the start of the window.

The audit focused on identifying authentication / authorization gaps, business-logic flaws, and implementation-level issues affecting the Dripster Leveraged Prediction Vault's off-chain control plane.

A total of 21 issues were identified and categorized as follows:

- 1 High severity
- 4 Medium severity
- 11 Low severity
- 5 Informational severity

Of the 21 findings, **13** were fixed during the engagement window itself and the remaining **8** were acknowledged with detailed rationale (route-disablement on dead surfaces, intended public API surfaces, test fixtures with no live secret material, or threat models superseded by the Polymarket V2 migration).

6.1 The Key Findings

- **State-Changing Operation via Unauthenticated GET:** A `GET /lending/referral-code/:walletAddress` route auto-created `ReferralCode` rows for any wallet address, violating HTTP GET safety and enabling unauthenticated bulk pre-seeding of codes for arbitrary addresses.
- **API Key Exposed in WebSocket Query Parameters:** Both authenticated WebSocket gateways accepted the partner API key via a `?apiKey=` handshake query string, leaking `dm_live_skey_…` credentials into access logs, reverse-proxy logs, and browser history. Public partner integration docs actively recommended the leaking form.

- **Cache Clear Endpoint Accepts Arbitrary Prefix Without Validation:** `DELETE /admin/v1/cache/:cachePrefix` forwarded an attacker-controlled URL path parameter into a Redis `SCAN MATCH` → `UNLINK` loop, allowing a compromised admin key to wipe security-critical prefixes such as `liquidationExecution`, `settlementExecution`, and `evmEventStreamDedup` — driving duplicate liquidations and on-chain event reprocessing.
- **Static Salt for API Key Hashing:** Every partner API key was hashed with a hardcoded checked-in salt, so a database dump alone was sufficient to mount a parallel brute-force against every API key hash. Resolved during the engagement by moving the static secret out of the database and into a server-side pepper held in GCP Secret Manager.

6.2 Architectural Security Observations

During the assessment, we observed several architectural decisions and controls that contribute to Dripster's security posture:

- **Hybrid On-Chain / Off-Chain Trust Model:** User collateral is locked on-chain while the backend holds the signing authority over every vault state transition — concentrating trust on the signing layer where hardware-backed protection (GCP KMS) is feasible.
- **Six-Layer Authorization Stack:** API key (PBKDF2 + HMAC pepper) → JWT with live DB revocation → role guards → object-level scoping → Prisma `WHERE`-clause ownership → on-chain GCP KMS signature. Compromise of one layer does not collapse the rest.
- **HSM-Backed Signing via GCP KMS:** Polygon vault transactions are signed inside a FIPS 140-2 Level 3 HSM; the application sends a digest and receives a signature. Even full RCE in the Cloud Run container cannot exfiltrate the signing key — the most valuable secret never exists in software memory.
- **Idempotent Lifecycle State Machine:** A BullMQ queue plus Redis-backed `AntiReplayStore` (`SET NX`) and a `DistributedMutexService` make every critical transition exactly-once at the operation level, with exponential-backoff retries and PagerDuty escalation on exhaustion.
- **Centralized Secret Management:** All production secrets are referenced by GCP Secret Manager resource name only. IAM-gated access plus Cloud Audit Logs on every secret read mean credential exfiltration requires IAM compromise rather than a leaked file.
- **Defense-in-Depth With Locatable Weaknesses:** Exceptional at the signing, async-state, and observability layers. Identified weaknesses are layer-specific and locally remediable (L-05 AES-CTR no auth tag; M-01 partner-admin object-level authorization) — neither requires structural redesign.

6.3 Security Strengths Observed

Based on our assessment, Dripster demonstrates the following security strengths:

- **HSM-Isolated Signing Authority:** All on-chain Polygon transactions are signed via GCP KMS (ECDSA P-256k1, FIPS 140-2 Level 3 HSM). The signing key never enters application memory, and every signing operation lands in Cloud Audit Logs with caller, timestamp, and key version.
- **Live JWT Revocation:** `CustomerJwtStrategy` hits the database on every request to confirm the embedded `apiKeyId` is still valid. Revoking a partner key invalidates every outstanding customer JWT immediately — a property stateless JWT systems rarely offer.
- **Idempotent Async State Machine:** Bull queue + Redis-backed `AntiReplayStore` pairs enforce exactly-once execution on financial operations. Double-settlement, double-liquidation, and replayed webhook events are blocked at the operation level even if they reach the queue multiple times.
- **Database-Level Ownership Enforcement:** Customer-facing position queries embed `walletAddress` and `partnerId` directly in the Prisma `where` clause. Even if app-level authorization were bypassed, the database returns no rows for another user's position.
- **Strong Webhook Verification:** QuickNode and Alchemy guards verify HMAC-SHA256 over the raw body via `timingSafeEqual`; QuickNode binds `nonce + timestamp + body` for replay resistance. The timestamp-window gap (L-07) was the only finding on the surface.
- **Adaptive ML-Driven Risk Engine:** An ONNX two-stage classifier sets per-market max leverage from order-book microstructure (spread, depth, top-holder concentration, order-flow imbalance) plus correlated crypto moves and Synthesis cross-validation — replacing static rules with adaptive exposure limits.

Summary of Findings:

Vulnerability	Severity	Status
[H-1] State-Changing Operation via GET: Unauthenticated Referral Code Auto-Creation	High	Acknowledged
[M-1] Potential IDOR: Cross-Partner API Key Management	Medium	Acknowledged
[M-2] API Key Exposed in WebSocket Query Parameters	Medium	Fixed
[M-3] Cache Clear Endpoint Accepts Arbitrary Prefix Without Validation	Medium	Fixed
[M-4] Telegram Webhook Controller Lacks Signature Verification	Medium	Acknowledged
[L-1] Overly Permissive CORS Policy	Low	Fixed
[L-2] Static Salt for API Key Hashing	Low	Fixed
[L-3] `bypassChecks` Flag Disables Risk Controls in Development	Low	Fixed
[L-4] Customer Swagger Docs Exposed in Production	Low	Acknowledged
[L-5] Consider replacing AES-256-CTR with AES-256-GCM	Low	Acknowledged
[L-6] Session Cookie Missing SameSite Attribute	Low	Fixed
[L-7] QuickNode Webhook Guard Lacks Timestamp Validation	Low	Fixed
[L-8] Session `saveUninitialized: true` Creates Sessions for Every Request	Low	Fixed
[L-9] Health Test Endpoints Leak Authentication Details in Development	Low	Acknowledged

Vulnerability	Severity	Status
[L-10] Request Body Logged in Plaintext for Slow Requests	Low	Fixed
[L-11] `$queryRawUnsafe` with Table Name String Interpolation in Watchdog Service	Low	Fixed
[I-1] Docker Compose: Hardcoded Credentials, No Redis Auth, Production Proxy Co-located	Info	Fixed
[I-2] Multiple Local Private Keys Hardcoded in .env.ci	Info	Acknowledged
[I-3] Privy JWT Private Key Committed to Repository in `.env.ci`	Info	Fixed
[I-4] Plaintext Production Polymarket Builder API Key in Committed `.env.prod`	Info	Acknowledged
[I-5] Gitleaks Explicitly Allowlists All Files Containing Secrets	Info	Fixed

7. Audit Findings

7.1 High Severity Findings

[H-01] State-Changing Operation via GET: Unauthenticated Referral Code Auto-Creation

Affected files

`src/lending/referral-code/referral-code.controller.ts#L11`
`src/lending/referral-code/referral-code.service.ts#L17`

CVSS 3.1

6.5 (Medium) — `CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:L`

Description

The `GET /lending/referral-code/:walletAddress` endpoint violates a fundamental HTTP safety requirement: GET requests must be safe and idempotent, they must not modify server state. This endpoint silently breaks that contract by writing a new database record every time it is called for a wallet address that does not yet have a referral code.

The logic inside `ReferralCodeService.getReferralCode()` first attempts a database lookup, and if the record is not found it immediately creates one rather than returning a 404:

```
// referral-code.service.ts
public async getReferralCode(walletAddress: string):
Promise<ReferralCode> {
  try {
    return await PrismaHelpers.execute(() =>
```

```
      this.prismaClient.referralCode.findFirstOrThrow({
        where: { walletAddress },
      }),
    );
  } catch (error) {
    if (error instanceof NotFoundException) {
      return this.createReferralCode(walletAddress); // ← DB
write on GET
    }
    throw error;
  }
}

private async createReferralCode(walletAddress: string):
Promise<ReferralCode> {
  const codeLength = 16;
  return PrismaHelpers.execute(() =>
    this.prismaClient.referralCode.create({
      data: {
        referralCodeId: ...,
        code: generateRandomString(codeLength),
        walletAddress,           // taken from caller-supplied
path param
        ...
      },
    }),
  );
}
```

This compounds into a more serious issue because the controller has no authentication guard applied at either the class or method level. The auto-creation path is therefore reachable by any HTTP client on the internet, including web crawlers, search engine indexers, browser link prefetchers, security scanners, and monitoring probes all of which routinely issue GET requests without credentials. None of these clients need any knowledge of the platform or its authentication model to trigger a database write.

Vulnerable Scenario:
The following steps help understand the issue:
1. A security scanner or web crawler discovers the endpoint URL pattern through Swagger exposure (the internal Swagger is publicly accessible on sandbox, see correspondent Finding), JavaScript bundle analysis, or network traffic interception.
2. The scanner begins issuing `GET /lending/referral-code/:address` requests against a list of wallet addresses sourced from public blockchain data.
3. For every wallet address in the list that has no existing referral code, the backend silently creates a new `ReferralCode` record in PostgreSQL and returns the generated code in the response.
4. The attacker now holds the referral codes for wallets they do not own, assigned before the legitimate users have ever interacted with the lending platform.
5. These pre-seeded codes may interfere with the legitimate onboarding flow if the platform uses a code's existence as a signal that a user has already been onboarded, or if the code is reused across systems.
6. At sufficient scale, this creates unbounded database growth as new `ReferralCode` rows are written for every enumerated address with no mechanism to clean them up.

Impact

Any unauthenticated HTTP client, including automated tools that would never reach an authenticated endpoint, can force database record creation for arbitrary wallet addresses by issuing a standard GET request. This enables bulk pre-seeding of referral codes across the entire addressable wallet population, interferes with legitimate onboarding flows, creates unbounded storage growth through automated enumeration, and exposes generated referral codes for wallets before their owners have interacted with the platform.

Recommendations

Separate the read and write operations. The GET handler should perform a pure lookup and return `404 Not Found` if no referral code exists for the target wallet.

Record creation must happen through an authenticated `POST` endpoint that verifies the caller owns the wallet.

Apply `CustomerJwtGuard` to the creation endpoint and derive the wallet address from the JWT's embedded `walletAddress` claim rather than accepting it as a URL parameter — this ensures a referral code can only be created by the authenticated wallet owner, not by any arbitrary caller supplying an address.

Status

Accepted by team — endpoint is fully disabled in every environment via `DeprecatedEndpointGuard` plus the `controllerBlocked` module flag (PR #3721).

7.2 Medium Severity Findings

[M-01] Potential IDOR: Cross-Partner API Key Management

Affected files

`src/common/services/partner/partner-admin.controller.ts#L74-L118`
`src/common/services/partner/partner.service.ts#L76-L88`

CVSS 3.1

6.5 (Medium) — `CVSS:3.1/AV:N/AC:L/PR:H/UI:N/S:U/C:H/I:H/A:N`

Description

The Partner Admin controller uses `AdminOrPartnerAdminApiKeyGuard` for API key lifecycle operations (create, list, delete). This guard allows both `admin` and `partnerAdmin` roles. However, the `partnerId` used in the operation is taken directly from the URL path parameter, there appears there is no validation that the authenticated `partnerAdmin`'s API key actually belongs to the target partner specified in the URL.

Vulnerable Code (controller — L74-L88):

```
// partner-admin.controller.ts L74-L88
@UseGuards(AdminOrPartnerAdminApiKeyGuard)  // <-- allows ANY
partnerAdmin
@HttpCode(HttpStatus.CREATED)
@Post(buildAdminPartnersApiKeysPath())
public async createApiKey(
  @Param(apiPathParam.partnerId) partnerId: string,  // <--
from URL, no ownership check
  @Body(new CamelizeBodyPipe()) body: CreatePartnerApiKeyBody,
): Promise<Decamelized<CreatePartnerApiKeyResult>> {
```

```
  try {
    const result = await
this.partnerService.createApiKeyForPartner({ partnerId, role:
body.role });
    //
^^^^^^^^^
    // partnerId from URL is passed directly, no check against
caller's partnerId
    return this.apiWrapperService.formatResponse(result);
  } catch (error) {
    return rethrowProperErrorForRestWithNotFound(error,
[PartnerNotFoundError], [PartnerApiKeyLimitExceededError]);
  }
}
```

Vulnerable Code (service — L76-L88):

```
// partner.service.ts L76-L88
public async createApiKeyForPartner(input:
CreatePartnerApiKeyInput): Promise<CreatePartnerApiKeyResult> {
  const partner = await
this.partnerRepository.findByPartnerId(input.partnerId);

  if (!partner) {
    throw new PartnerNotFoundError(input.partnerId);  // only
checks existence, NOT ownership
  }

  // ... creates key for the target partner and returns the
plaintext secret
  const apiKey = await this.apiKeyService.createApiKey({
partnerId: input.partnerId, role: input.role });
  return {
    id: apiKey.apiKeyId,
```

```
    publicId: apiKey.publicId,
    secretKey: apiKey.plainKey,  // <-- plaintext key returned
to attacker
    createdAt: new Date().toISOString(),
  };
}
```

The identical pattern exists at **L90-L103** (`listApiKeys`) and **L105-L118** (`deleteApiKey`).

Frontend / on-chain mitigation analysis: An admin dashboard frontend might only display the caller's own `partnerId` in the UI, making cross-partner requests less obvious. However, this is a direct API vulnerability, any `partnerAdmin` can craft an HTTP request with a different `partnerId` in the URL path.

Vulnerable Scenario:
1. Partner A has an API key with role `partnerAdmin` (apiKeyId: `ak_A`, partnerId: `par_A`).
2. Attacker authenticates with Partner A's `partnerAdmin` key.
3. Attacker calls `POST /admin/partners/par_B/api-keys` where `par_B` is a different partner.
4. The guard passes because the API key has role `partnerAdmin`.
5. `PartnerService.createApiKeyForPartner` only checks that `par_B` exists, not that the caller belongs to `par_B`.
6. A new API key is created for Partner B and **returned to the attacker in plaintext** (including the secret key).
7. Attacker now has API access to Partner B's account.

Impact

An attacker with `partnerAdmin` credentials for one partner might create, list, and delete API keys for every partner on the platform.

Recommendations

Inject the authenticated user's identity into the controller and validate that the `partnerId` in the URL matches the `partnerId` associated with the caller's API key. For example:

```
// Example code only DO NOT use it in PROD
@UseGuards(AdminOrPartnerAdminApiKeyGuard)
@Post(buildAdminPartnersApiKeysPath())
public async createApiKey(
  @Param(apiPathParam.partnerId) partnerId: string,
  @CurrentApiKey() currentApiKey: LoggedInApiKey,  // <-- add
this
  @Body(new CamelizeBodyPipe()) body: CreatePartnerApiKeyBody,
): Promise<...> {
  // Admin keys can manage any partner; partnerAdmin only their
own
  if (currentApiKey.role === ApiKeyRole.partnerAdmin
      && currentApiKey.partnerId !== partnerId) {
    throw new ForbiddenException();
  }
  // ...
}
```

Apply the same check to `listApiKeys` (L90) and `deleteApiKey` (L105).

Status

Acknowledged by team and accepted by auditor as by design. The team clarified that `partnerAdmin` is a platform-wide internal operator role (only issued to Dripster staff), not a tenant-scoped role assignable to external partners — so cross-partner management is the intended behavior. If `partnerAdmin` keys are ever issued to external partner companies, an ownership check at `AdminOrPartnerAdminApiKeyGuard` would become mandatory before that change ships.

[M-02] API Key Exposed in WebSocket Query Parameters

Affected files

`src/prediction-market/partner-position-websocket/partner-position-websocket.gateway.ts#L83-L97`

Description

The `PartnerPositionWebSocketGateway` accepts API keys via the `apiKey` query parameter in the WebSocket handshake URL. While the gateway also supports the `Authorization` header (L89-L94), the query parameter path is checked first and is the primary documented method (L84-L87).

Vulnerable Code (L83-L97):

```
// partner-position-websocket.gateway.ts L83-L97
private extractApiKey(client: Socket): string | undefined {
  const queryApiKey = client.handshake.query.apiKey;  // <--
checked first
  if (typeof queryApiKey === "string" && queryApiKey.length >
0) {
    return queryApiKey;  // API key in URL: wss://...?
apiKey=dm_live_skey_SECRET
  }

  const authHeader = client.handshake.headers.authorization;
  if (typeof authHeader === "string") {
    if (authHeader.startsWith(apiKeyAuthHeaderPrefix)) {
      return authHeader.slice(apiKeyAuthHeaderPrefix.length);
    }
  }
```

```
  return undefined;
}
```

Frontend / on-chain mitigation analysis: The frontend controls which authentication method is used. If the partner's frontend exclusively uses the `Authorization` header (L89-L94), no key appears in URLs. **However**, the backend actively supports and prioritizes the query parameter method, and partners integrating directly may default to this pattern since many WebSocket client libraries make query params easier than custom headers. The backend should enforce the secure path regardless of frontend behavior.

Impact

API key leakage through server logs, proxy logs, and browser history.

Recommendations

- Remove query parameter API key support entirely.
- Require API keys exclusively via the `Authorization` header.
- If query parameter auth is needed, implement a short-lived single-use ticket exchange.

Status

Fixed in PR #3685 — the query-string carrier was removed entirely; auth payload + `Authorization` header are now required.

[M-03] Cache Clear Endpoint Accepts Arbitrary Prefix Without Validation

Affected files

`src/common/services/cache/clear-cache.controller.ts#L14-L17`

`src/common/services/cache/clear-cache.service.ts#L17-L24`

`src/common/services/cache/cache-manager.service.ts#L61-L79`

`src/common/services/cache/base-cache.type.ts#L1-L55`

Description

The `DELETE /admin/v1/cache/:cachePrefix` endpoint takes a user-supplied URL parameter and performs a wildcard deletion across the entire Redis cache with zero validation or allowlisting.

At **clear-cache.controller.ts#L6-L17**, the controller is guarded only by `AdminApiKeyGuard` (role check) and passes the raw URL parameter directly to the service:

```
@UseGuards(AdminApiKeyGuard)
@UsePipes(new ValidationPipe({ transform: true }))
@Controller()
export class ClearCacheController {
  constructor(private readonly clearCacheService:
ClearCacheService) {}

  @UseGuards(AdminApiKeyGuard)
  @HttpCode(HttpStatus.ACCEPTED)
  @Delete(buildAdminApiPath(adminApiEndpoint.cache,
apiPathParam.cachePrefix))
  public async deleteCacheEntries(@Param("cachePrefix")
```

```
cachePrefix: string): Promise<void> {
    await
this.clearCacheService.deleteAllKeysStartsWith(cachePrefix);
// L16 — attacker-controlled prefix, no allowlist
  }
}
```

At **clear-cache.service.ts#L17-L24**, the service appends a colon and delegates to the cache manager:

```
public async deleteAllKeysStartsWith(prefix: string):
Promise<void> {
    if (!prefix) { return; }

    this.logger.log(`Deleting all cache keys starting with
${prefix}`);
    const count = await
this.cacheManagerService.deleteStartsWith(`${prefix}:`);  //
L23 — passes "${prefix}:*" to Redis SCAN
    this.logger.log(`Found and deleted ${count} keys starting
with ${prefix}`);
}
```

At **cache-manager.service.ts#L61-L79**, the Redis SCAN iterator matches all keys with the given prefix and unlinks them one by one:

```
public async deleteStartsWith(prefix: string): Promise<number>
{
    return this.getClient().execute(async (client) => {
      const stream = client.scanIterator({
        MATCH: `${prefix}*`,    // L65 — wildcard glob match on
user-controlled prefix
        COUNT: 100,
```

```
      });

      let count = 0;
      for await (const keysInBatch of stream) {
        for (const key of keysInBatch) {
          await client.unlink(key);  // L73 — deletes every
matching key
          count++;
        }
      }
      return count;
    });
}
```

The system defines **55 cache entry prefixes** at **base-cache.type.ts#L1-L55**, all of which are vulnerable to targeted deletion. The security-critical ones include:

```
export const CacheEntryPrefix = {
  cancelPositionAntiReplay: "cancelPositionAntiReplay",
// L4 — prevents duplicate position cancellations
  evmEventStreamDedup: "evmEventStreamDedup",
// L12 — deduplicates on-chain event processing
  evmNonce: "evmNonce",
// L14 — tracks EVM transaction nonces for the signing
authority
  liquidatablePositionsByMint: "liquidatablePositionsByMint",
// L27 — cached positions eligible for liquidation
  liquidationExecution: "liquidationExecution",
// L28 — prevents duplicate liquidation execution
  settlementExecution: "settlementExecution",
// L48 — prevents duplicate settlement execution
  unwindExecution: "unwindExecution",
// L53 — prevents duplicate force-unwind execution
```

```
  // ... 48 more prefixes
} as const;
```

Vulnerable Scenario:
1. An attacker obtains or compromises an admin API key (or a compromised insider with admin access acts maliciously).
2. The attacker calls `DELETE /admin/v1/cache/liquidationExecution`. This deletes the `LiquidationAntiReplayStore` entries, which use `executeOnlyOnce()` to prevent re-executing liquidations. With the anti-replay keys cleared, the next price check cycle will re-trigger liquidation flows for positions that have already been liquidated, potentially creating double-sell orders on Polymarket's CLOB and causing financial loss.

Impact

Financial Loss: Clearing `liquidationExecution` or `settlementExecution` anti-replay stores can trigger duplicate liquidation/settlement transactions, causing double-sell orders on Polymarket that result in direct financial loss from the protocol's capital pool.

State Corruption: Clearing `evmEventStreamDedup` causes reprocessing of on-chain events, potentially corrupting position state by applying the same state transition twice.

Recommendations

Even this has an admin guard consider Implementing a strict allowlist of cache prefixes that can be cleared (e.g., only non-critical caches like `offerExecutionData`, `inferenceMarketData`, `leverageThreshold`).

Add audit logging with alerting for all cache clear operations.

Consider requiring a confirmation token or two-step approval for destructive cache operations.

Status

Fixed by code removal in commit `f0379e821` (2026-04-18) — the controller, service, and route constants were all deleted.

[M-04] Telegram Webhook Controller Lacks Signature Verification

Affected files

`src/lending/telegram-bot/telegram-webhook-handler.controller.ts#L12, L27-L36`

`src/lending/telegram-bot/account/telegram-account.service.ts#L99-L107`

`src/common/providers/telegram/telegram-webhook.service.ts#L1-L40`

`src/common/providers/telegram/telegram-client.service.ts#L25-L36`

`src/common/providers/telegram/telegram-webhook.type.ts#L1-L62`

Description

The Telegram webhook controller is currently disabled by a hardcoded boolean, but when enabled it will accept arbitrary POST requests without any cryptographic verification of the request origin. Telegram's Bot API supports a `secret_token` parameter (set during `setWebhook`) that is sent in the `X-Telegram-Bot-Api-Secret-Token` header, this controller does not validate it.

At **telegram-webhook-handler.controller.ts#L11-L12**, the only protection is a compile-time constant:

```
// eslint-disable-next-line @typescript-eslint/consistent-type-assertions
const controllerBlocked = true as boolean;  // L12 - cast to boolean to prevent dead-code elimination, designed to be toggled
```

At **L14, L27-L36**, the controller has no `@UseGuards` decorator and no signature/token verification:

```
@Controller()
// L14 — no guard decorator
export class TelegramWebhookHandlerController {
  // ...

  @Post(buildLendingApiPath(lendingApiEndpoint.telegramWebhook,
undefined))          // L27 — publicly accessible POST endpoint
  public async processWebhook(@Body(new
ValidateAndStopAtFirstErrorPipe()) update: TelegramUpdate):
Promise<void> {
    if (controllerBlocked) {
      throw new ForbiddenException();
// L30 — only defense: a toggleable boolean
    }

    await this.telegramWebhookAntiReplayStore.executeOnlyOnce(
// L33 — anti-replay prevents duplicate update_id
      () => this.processTelegramUpdate(update),
//        but does NOT prevent forged payloads
      update.update_id.toString(),
// L35 — attacker controls update_id value
    );
  }
```

The anti-replay store at L33-L35 is keyed by `update_id`, which is an integer the attacker fully controls in the forged payload. By using unique `update_id` values, the attacker bypasses anti-replay entirely.

At **telegram-webhook-handler.controller.ts#L39-L52**, the handler parses the update and dispatches it:

```
private async processTelegramUpdate(update: TelegramUpdate):
Promise<void> {
    this.logger.log("Received Telegram webhook update", {
updateId: update.update_id });
    const parsedUpdate =
this.telegramWebhookService.parseUpdate(update);  // L42 –
parses forged payload

    if (!parsedUpdate) { return; }

    try {
      await
this.telegramAccountService.handleTelegramUpdate(parsedUpdate);
// L49 – executes account operations
    } catch (error) {
      this.logger.error("Error handling telegram update",
error);
    }
}
```

The `TelegramWebhookService.parseUpdate()` at **telegram-webhook.service.ts#L12-L36** extracts the command and arguments from the forged `message.text`:

```
public parseUpdate(update: TelegramUpdate):
ProcessedTelegramUpdate | null {
    if (!update.message?.text) { return null; }

    const message = update.message;
    const text = message.text;
    const isCommand = !!text?.startsWith("/");

    let command: string | undefined;
    let commandArgument: string | undefined;
```

```
    if (text && isCommand) {
      const parts = text.split(" ");
      const firstPart = parts[0];
      if (firstPart) {
        command = firstPart.substring(1);     // L28 –
attacker-controlled command name
      }
      commandArgument = parts.slice(1).join(" ");  // L30 –
attacker-controlled argument
    }

    return {
      chatId: message.chat.id,       // attacker-controlled
chat ID
      username: message.from.username, // attacker-controlled
username
      text,
      isCommand,
      command,
      commandArgument,
    };
}
```

The `TelegramAccountService.handleTelegramUpdate()` at **telegram-account.service.ts#L99-L107** processes the `/start` command to link Telegram accounts:

```
public async handleTelegramUpdate(update:
ProcessedTelegramUpdate): Promise<void> {
    if (!update.isCommand || !update.command) { return; }

    if (update.command === "start" && update.commandArgument) {
      await this.linkAccount(update.chatId,
```

```
update.commandArgument, update.username);  // L105 — links
attacker's chatId using forged authCode
    }
}
```

The `linkAccount` method at **telegram-account.service.ts#L53-L89** validates the auth code against the database, then creates the account link and sends a confirmation message through the bot:

```
public async linkAccount(chatId: number, authCode: string,
username?: string): Promise<TelegramAccount> {
    const authCodeRecord = await this.findAuthCode(authCode);
    if (!authCodeRecord) { throw new
TelegramAuthCodeInvalidError(authCode); }
    if (authCodeRecord.expiresAt < new Date()) { throw new
TelegramAuthCodeExpiredError(authCode); }

    // ... creates TelegramAccount record with attacker's
chatId ...

    await this.telegramClientService.sendMessage(       // L83
— sends confirmation to attacker-controlled chatId
      chatId,
      "Your Telegram account has been successfully linked! You
will now receive liquidation alerts.",
    );
    return account;
}
```

The `TelegramClientService.sendMessage()` at **telegram-client.service.ts#L25-L36** sends messages using the real bot API token:

```
public async sendMessage(chatId: number | string, text: string,
parseMode?: "HTML" | "Markdown"): Promise<void> {
    const payload: TelegramSendMessageParams = { chat_id:
chatId, text };
    // ...
    const response = await this.axiosService.post(new
URL(`${this.baseUrl}/sendMessage`), payload);  // L36 — sends
via real bot token
}
```

Vulnerable Scenario:

1. The `controllerBlocked` constant is toggled to `false` (it is explicitly designed for this, the `as boolean` cast prevents TypeScript dead-code elimination, indicating the team intends to enable it).
2. An attacker discovers the webhook URL (e.g., via the Swagger documentation exposed in production, or by guessing the lending API path convention).
3. The attacker sends a forged POST request:

```
{
  "update_id": 999999,
  "message": {
    "message_id": 1,
    "from": { "id": 12345, "is_bot": false, "first_name":
"Attacker", "username": "attacker" },
    "chat": { "id": 67890, "type": "private" },
    "date": 1700000000,
    "text": "/start VALID_AUTH_CODE_HERE"
  }
}
```

If the attacker has intercepted or guessed a valid auth code (16-character random string, valid for 30 minutes per `telegram-account.service.ts#L36-L37`), the

`linkAccount` method binds the attacker's `chatId` (67890) to the victim's wallet address.

Even without a valid auth code, the attacker can enumerate auth codes via brute-force, the endpoint has no rate limiting (`@Throttle` decorator is absent), and the error responses differentiate between "invalid" and "expired" codes (`TelegramAuthCodeInvalidError` vs `TelegramAuthCodeExpiredError`), enabling timing-based enumeration.

Impact

Account Hijacking: An attacker can link their Telegram account to a victim's wallet address by forging a webhook payload with a valid auth code, intercepting all future liquidation alerts and position notifications.

Information Disclosure: Liquidation alerts contain position-specific data (prices, leverage levels, market tickers) that enables front-running or informed trading against the victim's positions.

Auth Code Enumeration: The absence of rate limiting on the webhook endpoint and differentiated error responses enable brute-force enumeration of active auth codes within their 30-minute validity window.

Recommendations

Implement Telegram's webhook secret token verification: set a `secret_token` during the `setWebhook` API call and validate the `X-Telegram-Bot-Api-Secret-Token` header in a guard before processing the payload.

Add rate limiting (`@Throttle`) to the webhook endpoint.

Status

Accepted by team — endpoint is double-blocked in every environment via `DeprecatedEndpointGuard` and a module-scope `controllerBlocked` flag (PR #3721).

7.3 Low Severity Findings

[L-01] Overly Permissive CORS Policy

Affected files

`src/common/runtime/bootstrap.utils.ts#L134-L140`

Description

Vulnerable Code (L134-L140):

```
// bootstrap.utils.ts L134-L140
function initCors(app: NestExpressApplication): void {
  app.enableCors({
    origin: true,         // <-- reflects ALL origins
    methods: "GET,HEAD,PUT,PATCH,POST,DELETE,OPTIONS",
    credentials: true,    // <-- allows cookies cross-origin
  });
}
```

Frontend / on-chain mitigation analysis: CORS is a server-side policy, no frontend or on-chain control can mitigate this. The primary customer API authentication uses `Authorization` headers (not automatically attached cross-origin by browsers), reducing practical CSRF risk.

Impact

Cross-site request forgery and data exfiltration for cookie-authenticated endpoints. Practical impact is currently limited because the primary auth uses `Authorization` headers.

Recommendations

Replace `origin: true` with an explicit allowlist of trusted origins.

Status

Fixed by infrastructure deletion in PR #3706 — the dead session stack (`express-session`, `cookie-parser`, `connect-redis`) was removed entirely and CORS is now `{ origin: true, credentials: false }`.

[L-02] Static Salt for API Key Hashing

Affected files

src/common/auth/api-key/api-key.service.ts#L12
src/common/auth/api-key/api-key.service.ts#L36-L38
src/common/services/hashing/hashing.service.ts#L10-L18

Description

Vulnerable Code (api-key.service.ts L12, L36-L38):

```
// api-key.service.ts L12
const apiKeyHashingSalt = "ipump";  // <-- hardcoded, same for
ALL keys

// api-key.service.ts L36-L38
public async getByKey(key: string): Promise<ApiKey> {
  const hashedKey = this.hashingService.hash(key,
apiKeyHashingSalt);  // static salt used
  return this.apiKeyRepository.findByHash(hashedKey.hash);
}
```

Vulnerable Code (hashing.service.ts L10-L18):

```
// hashing.service.ts L10-L18
public hash(input: string, salt: string | undefined):
HashResult {
  const iterations = 100000;
  const keyLength = 64;
  const digest = "sha512";
  const saltLength = 32;
  const saltOrRandomSalt = salt ??
```

```
randomBytes(saltLength).toString("hex");
  //                          ^^^^
  // When salt is provided (always "ipump"), no random salt is
generated
  const hash = pbkdf2Sync(input, saltOrRandomSalt, iterations,
keyLength, digest).toString("hex");
  return { hash, salt: saltOrRandomSalt };
}
```

Frontend / on-chain mitigation analysis: Not applicable, purely backend cryptographic concern.

Impact

If the database is compromised, the attacker knows the salt for every API key hash, enabling parallel brute-force against all hashes simultaneously. High key entropy (64 hex chars) provides significant resistance, but the static salt weakens defense-in-depth.

Recommendations

Use the per-key salt already stored in the database.

Status

Fixed in PR #3687 — added a server-side pepper from GCP Secret Manager (`HMAC-SHA256(PBKDF2(plainKey, salt), pepper)`); a database dump alone no longer compromises the hashes.

[L-03] `bypassChecks` Flag Disables Risk Controls in Development

Affected files

src/prediction-market/offer/offer.controller.ts#L100-L103
src/prediction-market/offer/offer.service.ts#L69

Description

Vulnerable Code (offer.controller.ts L99-L103):

```
// offer.controller.ts L99-L103
const isSandbox =
this.sandboxService.isSandboxRole(customer.role);
const bypassChecks = isSandbox || (query.bypassChecks ??
false);
if (bypassChecks && !isSandbox &&
this.environmentService.isInProdEnvironment()) {
  throw new ForbiddenException("bypassChecks is not allowed in
production");
}
// In dev: any customer JWT holder can bypass ALL risk checks
```

Vulnerable Code (offer.service.ts L69):

```
// offer.service.ts L69
const bypassChecks = input.isSandbox || (input.bypassChecks &&
!this.environmentService.isInProdEnvironment());
// When bypassChecks is true, the following are skipped:
//   - Position limits (global, market, partner, user, side) -
offer.service.ts ~L110
//   - Slippage validation - offer.service.ts ~L118
//   - Entry filters (depth, spread, volume, staleness) -
```

```
offer.service.ts ~L122
//   - Leverage clamping from risk model — offer.service.ts
~L191
//   - Market eligibility checks — offer.service.ts ~L139
```

Frontend / on-chain mitigation analysis: This is just a dev sandbox`bypassChecks=true` of course. However, it's a documented query parameter in `offer.args.ts` and any direct API caller can include it. The **on-chain contract enforces its own invariants** (collateral requirements, signature expiry, state machine) regardless of `bypassChecks`, providing partial mitigation. But the risk engine's slippage, leverage, and market eligibility checks exist precisely because on-chain invariants alone are insufficient to prevent bad trades, the backend signs offers that the contract trusts, so a badly-checked offer translates to real financial exposure.

Impact

Complete bypass of the risk engine in non-production environments. If dev infrastructure shares signing keys or CLOB accounts with production, this enables unrestricted leveraged trading.

Recommendations

Restrict `bypassChecks` to admin API keys only.

Even in dev, apply minimum safety guards (notional caps).

Status

Resolved — the `bypassChecks` query parameter and the `isSandbox` bypass branch have been removed from offer creation; every environment now enforces the full check set.

[L-04] Customer Swagger Docs Exposed in Production

Affected files

`src/common/runtime/bootstrap.utils.ts#L164-L167`

Description

Vulnerable Code (L164-L167):

```
// bootstrap.utils.ts L164-L167
if (!environmentService.isInProdEnvironment()) {
  initSwagger(app);         // Internal swagger, gated ✓
}
initCustomerSwagger(app);   // Customer swagger, ALWAYS
exposed, including production
```

Frontend / on-chain mitigation analysis: The Swagger UI at `/v1/customer-docs` provides a complete endpoint map to attackers. Considering the customer API is designed for B2B partner consumption, this may be intentional, but should be verified.

Recommendations

If intentional, ensure no internal endpoints leak through module imports. If not, gate behind the same environment check.

Status

Accepted by team — intentional public API surface for partner client generation; CI gates already prevent admin/oracle/lending/webhook paths from leaking into the customer spec.

[L-05] Consider replacing AES-256-CTR with AES-256-GCM

Affected files

`src/common/services/encryption/encryption.service.ts#L6-L34`

`src/common/services/partner/partner.service.ts#L156-L165`

Description

The `EncryptionService` uses AES-256-CTR mode with an initialization vector (IV) derived deterministically from the salt parameter via `MD5(salt)`.

The cipher mode and IV derivation at **encryption.service.ts#L6, L24, L27, L32-L34**:

```
const encryptionMethod = "aes-256-ctr";  // L6 - CTR mode
requires unique IV per encryption

public encryptString(secretValue: string, salt: string | null,
keyName: keyof EncryptionKeys): string {
    const saltBuffer = this.getHashedSaltBuffer(salt ??
keyName);  // L24 - IV derived from salt; if null, falls back
to keyName string

    const key = this.encryptionKey[keyName];
    const cipher = createCipheriv(encryptionMethod, key,
saltBuffer);  // L27 - deterministic IV reused across calls
with same salt+key

    return Buffer.concat([cipher.update(secretValue),
cipher.final()]).toString("hex");
}

private getHashedSaltBuffer(salt: string): Buffer {
```

```
    return createHash("md5").update(salt).digest();  // L33 –
MD5 of salt = 16 bytes = AES-CTR IV. Same input always yields
same IV.
}
```

The call site at **partner.service.ts#L156-L165** encrypts two different secrets (passphrase and secret) for the same partner with the identical salt (`partnerId`) and key (`"polymarketBuilderEncryptionKey"`):

```
polymarketBuilderApiPassphraseEncrypted:
this.encryptionService.encryptString(
    builder.passphrase,
    partnerId,                          // L158 – salt =
partnerId
    "polymarketBuilderEncryptionKey",   // L159 – same key name
),
polymarketBuilderApiSecretEncrypted:
this.encryptionService.encryptString(
    builder.secret,
    partnerId,                          // L163 – salt =
partnerId (identical to above)
    "polymarketBuilderEncryptionKey",   // L164 – same key name
(identical to above)
),
```

Vulnerable Scenario:

1. Both `builder.passphrase` and `builder.secret` are encrypted under the same AES key with `IV = MD5(partnerId)`.
2. AES-CTR with a reused key+IV pair means `ciphertext_A XOR ciphertext_B = plaintext_A XOR plaintext_B`.
3. An attacker with a potential database read access (SQL injection, backup leak, compromised cloud IAM) XORs the two ciphertexts.
4. If one plaintext is known or partially guessable, the other is fully recoverable.

5. This pattern extends to every `EncryptionKeys` member used with the same salt, faker private keys, oracle signing keys, Solana authority keys, trading bot keys, etc.

Impact

While the attack complexity is evidently considerable a potential attacker with database read access can decrypt all secrets without needing the encryption keys themselves.

Recommendations

Replace AES-256-CTR with AES-256-GCM (authenticated encryption).

Generate a cryptographically random 96-bit IV per encryption operation and prepend it to the ciphertext.

Never derive IVs deterministically from predictable inputs.

Status

Accepted by team — the specific reuse pattern is gone after the Polymarket V2 migration removed per-partner builder credentials, and remaining `EncryptionService` consumers each use a distinct `keyName` with a single secret per (salt, keyName) pair.

[L-06] Session Cookie Missing SameSite Attribute

Affected files

`src/common/runtime/bootstrap.utils.ts#L56-L72`

Description

The session cookie configuration does not set `sameSite`.

At **bootstrap.utils.ts#L56-L72**:

```
app.use(
    session({
      cookie: {
        domain,
        httpOnly: true,
        maxAge: thirtyMinutes,
        path: "/",
        secure: environmentService.isInDevOrProdEnvironment(),
        // NOTE: no sameSite attribute configured anywhere in
this object
      },
      proxy: true,
      resave: false,
      rolling: true,
      saveUninitialized: true,
      secret: cookieSignatureSecret,
      store: new RedisStore({ client: redisClient }),
    }),
);
```

The `cookie` object at L58-L64 sets `httpOnly`, `secure`, `domain`, `maxAge`, and `path`, but omits `sameSite` entirely.

Impact

Potential CSRF attacks depending on browser defaults.

Recommendations

- Add `sameSite: 'strict'` (or `'lax'`) to the cookie configuration.

Status

Fixed alongside L-01 in PR #3706 — the entire session stack was deleted, so no cookie is ever issued and the missing `sameSite` attribute is vacuous.

[L-07] QuickNode Webhook Guard Lacks Timestamp Validation

Affected files

`src/common/providers/quicknode/quicknode-webhook-signature.guard.ts#L21-L50`

Description

The guard verifies the HMAC signature but does not validate the `timestamp` header against a time window.

At **quicknode-webhook-signature.guard.ts#L29-L47**:

```
const signature = request.headers[quicknodeSignatureHeader];
const nonce = request.headers[quicknodeNonceHeader];
const timestamp = request.headers[quicknodeTimestampHeader];
// L31 - timestamp extracted

if (typeof signature !== "string") { throw new
UnauthorizedException(...); }
if (typeof nonce !== "string" || typeof timestamp !== "string")
{ throw new UnauthorizedException(...); }

const expectedSignature = createHmac("sha256",
this.quicknodeConfig.webhookSigningKey)
    .update(nonce)
    .update(timestamp)    // L43 - timestamp included in HMAC
but never validated for recency
    .update(rawBody)
    .digest("hex");

assertWebhookSignaturesMatch(expectedSignature, signature);  //
```

```
L47 — signature valid, but could be hours/days old
return true;
```

No check like `if (Math.abs(Date.now() - parseInt(timestamp)) > maxAge) throw ...` exists. A captured valid payload can be replayed indefinitely.

Recommendations

> Validate that the timestamp is within an acceptable window (e.g., ±300 seconds).

Status

Fixed in PR #3742 — added a ±300 s symmetric timestamp window check that runs before the HMAC computation.

[L-08] Session `saveUninitialized: true` Creates Sessions for Every Request

Affected files

`src/common/runtime/bootstrap.utils.ts#L68`

Description

At **bootstrap.utils.ts#L68**:

```
saveUninitialized: true,  // L68 - creates and persists a new
Redis session for EVERY incoming request
```

This creates empty session entries in Redis for webhook traffic, health probes, and unauthenticated requests.

Recommendations

Set `saveUninitialized: false`.

Status

Fixed alongside L-01 in PR #3706 — the entire session stack was deleted, so no records are written to Redis at all.

[L-09] Health Test Endpoints Leak Authentication Details in Development

Affected files

`src/common/controllers/health/health.controller.ts#L86-L116`

Description

At **health.controller.ts#L89-L91** and **L113-L115**:

```
public testAdminAuth(@CurrentApiKey() apiKey: LoggedInApiKey):
Record<string, unknown> {
    this.assertNotProduction();
    return { authType: "admin", apiKeyId: apiKey.apiKeyId,
partnerId: apiKey.partnerId };  // L91 - echoes back
credentials
}
```

```
public testCustomerAuth(@CurrentCustomer() customer:
LoggedInCustomer): Record<string, unknown> {
    this.assertNotProduction();
    return { authType: "customer", walletAddress:
customer.walletAddress, partnerId: customer.partnerId };  //
L115
}
```

These endpoints return API key IDs, partner IDs, and wallet addresses, useful for enumeration attacks in the dev environment.

Recommendations

Return only a boolean success indicator, not authentication details.

Status

Accepted by team — endpoints are prod-disabled via `assertNotProduction()` and only echo the caller's own credential identity; the returned fields are load-bearing test observables in `health-auth.e2e.ts` and `throttling.e2e.ts`.

[L-10] Request Body Logged in Plaintext for Slow Requests

Affected files

`src/common/api/request/logging.interceptor.ts#L85-L91`

CVSS 3.1

6.5 (Medium) — `CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N`

CVSS 3.1

8.7 (High) — `CVSS:3.1/AV:N/AC:L/PR:H/UI:N/S:C/C:N/I:H/A:H`

CVSS 3.1

6.5 (Medium) — `CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N`

CVSS 3.1

3.1 (Low) — `CVSS:3.1/AV:N/AC:H/PR:N/UI:R/S:U/C:N/I:L/A:N`

CVSS 3.1

2.2 (Low) — `CVSS:3.1/AV:N/AC:H/PR:H/UI:N/S:U/C:L/I:N/A:N`

CVSS 3.1

3.1 (Low) — `CVSS:3.1/AV:N/AC:H/PR:L/UI:N/S:U/C:N/I:L/A:N`

CVSS 3.1

3.7 (Low) — `CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N`

CVSS 3.1

1.9 (Low) — `CVSS:3.1/AV:L/AC:H/PR:H/UI:N/S:U/C:L/I:N/A:N`

CVSS 3.1

3.1 (Low) — `CVSS:3.1/AV:N/AC:H/PR:N/UI:R/S:U/C:N/I:L/A:N`

CVSS 3.1

3.7 (Low) — `CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N`

CVSS 3.1

3.7 (Low) — `CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:L`

CVSS 3.1

3.3 (Low) — `CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N`

CVSS 3.1

2.3 (Low) — `CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:U/C:L/I:N/A:N`

Description

The `LoggingInterceptor` logs the full request body as a serialized JSON string for any request that exceeds its duration threshold:

```
this.logger.warn(
  `Long running request of ${requestDuration}ms on
${handlerName}. ` +
    `Method: ${request.method} URL: ${request.url} Body:
${JSON.stringify(request.body)} ` +
    `Query: ${JSON.stringify(request.query)}`,
);
```

For the offer creation endpoint (`createOffer`/`createQuote`), the threshold is 5 seconds. A slow offer request would log the entire request body including `leverageBps`, `marketTicker`, `notionalAmountUsdPips`, `slippageBps`, and `effectiveSide`, a complete description of the user's trading intent. This data is written to Google Cloud Logging, Sentry, and potentially third-party log aggregation services, where it persists and may be accessible to a broader set of personnel or systems than the financial trading data itself warrants.

Vulnerable Scenario:
The following steps help understand the issue:
1. An authenticated user submits a large leveraged offer that takes more than 5 seconds to process.
2. The logging interceptor fires and writes the complete body, leverage, market ticker, position size, slippage tolerance to the server log.
3. A GCP Logging access credential is compromised.
4. The attacker now knows the exact trading parameters for high-value positions, targeted liquidation attacks, or leakage of proprietary trading strategy.

Impact

Sensitive trading strategy parameters (leverage, notional size, target market, slippage tolerance) are written to server logs whenever requests are slow, creating a data retention risk and potential information disclosure to anyone with log access.

Recommendations

Remove or redact the `Body: ${JSON.stringify(request.body)}` component from the slow-request log message entirely.

If body logging is required for debugging, apply an explicit allowlist of non-sensitive fields (e.g., only `pmProvider`) or hash/mask financial values.

Ensure log access is governed by the same access controls as trading data.

Status

Fixed by code change — the `Body:` and `Query:` segments were removed from the slow-request log line; method, URL, handler, and duration are retained for triage.

[L-11] `$queryRawUnsafe` with Table Name String Interpolation in Watchdog Service

Affected files

`src/oracle/prediction-market/prediction-market-watchdog/prediction-market-watchdog.service.ts#L216-L224`

CVSS 3.1

2.2 (Low) — `CVSS:3.1/AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N`

Description

The `getLastComputedFeedPublishedDate` method uses Prisma's `$queryRawUnsafe`, the explicitly unsafe variant that does not parameterize inputs, with a `tableName` variable interpolated directly into the SQL string:

```
const tableName = feedType;
const result = await this.prismaClient.$queryRawUnsafe<{
publishedAt: Date }[]>(
  `SELECT t."publishedAt" FROM "Market" m INNER JOIN
"${tableName}" t ON t."marketId" = m.id ...`,
  marketId,
);
```

A whitelist check (`knownFeedTypes.includes(feedType)`) runs immediately before this call and currently limits `feedType` to three known table names. However:

> The function uses `$queryRawUnsafe` rather than the safe `$queryRaw` with `Prisma.raw`. This means if the whitelist check is ever weakened, removed, or

bypassed, the raw string interpolation directly enables SQL injection.

`feedType` originates from a raw database query result (`row.feedType` from `$queryRaw`), which is typically safe but creates a trust chain that is difficult to audit: if the upstream raw query ever returns a manipulated value, it flows directly into an unsafe SQL string.

Impact

Currently mitigated by the whitelist, but the use of `$queryRawUnsafe` with string interpolation creates a latent SQL injection risk that could be triggered by future code changes.

Recommendations

Replace `$queryRawUnsafe` with the safe `$queryRaw` using `Prisma.raw` for the table name component:

```
const result = await this.prismaClient.$queryRaw<{ publishedAt:
Date }[]>(
  Prisma.sql`SELECT t."publishedAt" FROM "Market" m INNER JOIN
${Prisma.raw(`"${tableName}"`)} t ON t."marketId" = m.id WHERE
m."marketId" = ${marketId} ORDER BY t."publishedAt" DESC LIMIT
1`,
);
```

Retain the whitelist as defense-in-depth.

Status

Fixed in PR #3745 — `$queryRawUnsafe` was replaced with `$queryRaw` + `Prisma.sql`, and now appears in zero production call sites.

7.4 Informational Severity Findings

[I-01] Docker Compose: Hardcoded Credentials, No Redis Auth, Production Proxy Co-located

Affected files

`docker-compose.yml#L12-L13, #L29-L33, #L74-L87`

Description

```
# L12-L13 - hardcoded Postgres password in version control
POSTGRES_PASSWORD: lentille-de-fresnel

# L29-L33 - Redis with no authentication, bound to 0.0.0.0
redis:
  ports:
    - "6379:6379"

# L74-L87 - PRODUCTION read-replica proxy in local dev compose
gcp-sql-proxy-prod-replica:
  volumes:
    - ./credentials/dripster-backend-prod.json:/config
```

Recommendations

Move production proxy to a separate compose file.

Add Redis `requirepass`. Bind to `127.0.0.1`.

Status

Partially fixed in PR #3746 — prod read-replica and sandbox SQL proxies were moved to a separate `docker-compose.remote-db.yml` and bound to `127.0.0.1` only. Local-dev Postgres password and Redis no-auth were left as-is by team decision (no LAN exposure, no prod data).

[I-02] Multiple Local Private Keys Hardcoded in .env.ci

Affected files

`.env.ci#L4–L39`

CVSS 3.1

Informational — no CVSS score applicable (defensive observation, no exploitable vector at the time of testing).

CVSS 3.1

Informational — no CVSS score applicable (defensive observation, no exploitable vector at the time of testing).

Description

The CI environment file, which is committed to the repository and therefore present in the full Git history, contains multiple private keys in plaintext.

While labeled "local", if any of these keys correspond to wallets used in shared environments, or if they control any on-chain assets (even testnet), the risk is immediate.

- `FREE_MINTS_LOCAL_PRIVATE_KEY` — Solana wallet private key (Base58, 64 bytes)
- `PM_SOLANA_AUTHORITY_LOCAL_PRIVATE_KEY` — Solana PM authority key
- `PM_EVM_FAKER_LOCAL_PRIVATE_KEY=0x7c852118...` — Ethereum private key

`POLYMARKET_ORDER_FAKER_LOCAL_PRIVATE_KEY=0x47e179ec...` — Ethereum private key

`PRICE_SIMULATOR_LOCAL_PRIVATE_KEY` — Solana key

`DFLOW_ORDER_FAKER_LOCAL_PRIVATE_KEY` — Solana key

`PM_FAKER_LOCAL_PRIVATE_KEY` — Solana key

This is a finding just to ack their presence, independently of the usage which is why it is labeled as INFO

Impact

Complete key compromise for all accounts listed. If any of these wallets hold assets or have signing authority in any non-fully-isolated environment, funds and signed data are at risk.

Recommendations

Remove all private key material from `.env.ci` immediately.

Add `gitleaks` or `truffleHog` as a mandatory pre-commit hook and CI gate. The repository already has `.gitleaks.toml`, but CI enforcement appears absent.

Status

Acknowledged by team and accepted by auditor as test fixtures with no live secret material. The EVM keys are Anvil default-mnemonic accounts (public to every Foundry/Hardhat

install), and every `_LOCAL_PRIVATE_KEY` binds only to faker / simulator services — prod analogues come from GCP Secret Manager.

[I-03] Privy JWT Private Key Committed to Repository in `.env.ci`

Affected files

`.env.ci#L9-L20`

CVSS 3.1

Informational — no CVSS score applicable (defensive observation, no exploitable vector at the time of testing).

Description

The CI environment file contains a full PEM-encoded ECDSA private key (`LOCAL_PRIVY_JWT_PRIVATE_KEY`) used to sign Privy JWTs. This is a separate finding from the wallet keys above because JWT signing keys carry a distinct threat: possession of this key allows an attacker to potentially mint valid JWTs that impersonate any wallet address, bypassing all API authentication. The corresponding public key is also committed as `LOCAL_PRIVY_JWT_PUBLIC_KEY`.

This is a finding just to ack their presence, independently of the usage which is why it is labeled as INFO

Impact

Ability to forge authentication tokens for any wallet address, granting full API access as any user without legitimate credentials.

Recommendations

Immediately rotate the exposed ECDSA key pair and re-issue any dependent credentials.

Verify that `LOCAL_PRIVY_JWT_PUBLIC_KEY` is not registered or trusted in any live environment.

Status

Fixed in PR #3748 — the Privy JWT keypair was a leftover from a decommissioned integration with no live consumer, and was deleted outright (env files + env-var-name registry).

[I-04] Plaintext Production Polymarket Builder API Key in Committed `.env.prod`

Affected files

`.env.prod#L56`

CVSS 3.1

Informational — no CVSS score applicable (defensive observation, no exploitable vector at the time of testing).

Description

Every other secret in `.env.prod` is correctly stored in GCP Secret Manager and referenced by resource name. A single entry is an exception, the actual secret value of the Polymarket Builder API key is committed directly in plaintext:

```
POLYMARKET_BUILDER_API_KEY=019d77cc-e374-7036-b843-185b5555df2e
```

This key controls Polymarket Builder API authentication and is a live production credential. Any party with repository read access can extract it without requiring GCP IAM access.

Impact

Unauthorized control of Bloom's production Polymarket Builder identity, enabling order placement, modification, or cancellation on live markets.

Recommendations

Immediately rotate the key `019d77cc-e374-7036-b843-185b5555df2e` with Polymarket.

Move the secret to GCP Secret Manager as `POLYMARKET_BUILDER_API_KEY_SECRET_NAME`, consistent with every other production secret in the same file.

Status

Acknowledged by team and accepted by auditor — `POLYMARKET_BUILDER_API_KEY` is Polymarket's public builder *code* (not a credential), and was already removed from `.env.prod` as part of the V2 migration.

[I-05] Gitleaks Explicitly Allowlists All Files Containing Secrets

Affected files

`.gitleaks.toml#L7-L33`

CVSS 3.1

Informational — no CVSS score applicable (defensive observation, no exploitable vector at the time of testing).

Description

The repository ships a Gitleaks configuration, suggesting active intent to prevent secret leakage. However, the `[allowlist]` section explicitly exempts precisely the files that contain the most critical secrets:

```
paths = [
  '''\\.env\\.ci$''',
  '''\\.env\\.local\\.sample$''',
  '''\\.env\\.dev$''',
  '''\\.env\\.prod$''',
  '''\\.mock\\.ts$''',
  '''\\.stub\\.ts$''',
  ...
]
```

We understand dev acks this but this still should be reported as INFORMATIONAL only.

Recommendations

Remove `.env.ci`, `.env.local.sample`, `.env.dev`, and `.env.prod` from the Gitleaks allowlist paths immediately.

Enforce Gitleaks as a blocking CI check on all branches, not just `main`.

Status

Fixed in PR #3748 — env files were removed from the `.gitleaks.toml` `paths` allowlist and per-line `# gitleaks:allow` annotations were added to known-safe fixtures only.