CD SECURITY

AUDIT REPORT

Cerebro

December 2025

Introduction

A time-boxed security review of the **Cerebro** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A code review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Cerebro**

Cerebro is a Web3 portfolio-management and DeFi/NFT dashboard that aggregates holdings across multiple blockchains and wallets. It offers real-time tracking of token balances (fungible and non-fungible), net-worth calculation, cost-basis and yield monitoring, liquidity and allocation analytics.

Severity classification

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

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 8a282dfc6164ee72cb6136e8727051b30363ccfe**

Scope

The following smart contracts were in scope of the audit:

- `apps/backend/src/auth/auth.controller.ts`
- `apps/backend/src/auth/auth.service.ts`
- `apps/backend/src/auth/facade/auth-facade.service.ts`

- `apps/backend/src/auth/services/two-factor-check.service.ts`
- `apps/backend/src/wallet-accounts/wallet-accounts.controller.ts`
- `apps/backend/src/wallet-accounts/wallet-accounts.service.ts`
- `apps/web/providers/privy-provider.tsx`
- `apps/backend/src/swap/swap.service.ts`
- `apps/web/components/swap/*`
- `apps/web/lib/fetchers/swap.ts`
- `apps/web/lib/swap/execute-privy-solana-transaction.ts`
- `apps/web/lib/swap/execute-solana-transaction.ts`
- `apps/web/lib/swap/execute-transaction.ts`

The following number of issues were found, categorized by their severity:

- Critical & High: 6 issues
- Medium: 6 issues
- Low: 7 issues

Findings Summary

ID	Title	Severity	Status
[H-01]	Privilege Escalation - wallet verified flag	High	Fixed
[H-02]	State changing GET and sensitive token exposure in body	High	Fixed
[H-03]	CSRF risk due to cookie based JWT with SameSite=None	High	Fixed
[H-04]	NoSQL/MongoDB operator injection in update endpoints	High	Fixed
[H-05]	Second-order denial of service	High	Fixed
[H-06]	Broken Access Control allowing Waitlist Bypass	High	Fixed
[M-01]	WRace Condition - wallet subscription limit	Medium	Fixed
[M-02]	Unbounded credentials length enables DoS	Medium	Fixed
[M-03]	CSRF and abuse on public refresh-token rotation	Medium	Fixed
[M-04]	Swap history manipulation	Medium	Fixed
[M-05]	Authentication Bypass of 2FA via Mobile Platform Flag	Medium	Fixed
[M-06]	Missing Rate Limiting allows Referral Code Bruteforcing	Medium	Fixed
[L-01]	Precision Loss in Token Amount Parsing	Low	Fixed
[L-02]	Raw Error Message Exposure	Low	Fixed
[L-03]	Missing HTTP Security Headers	Low	Fixed
[L-04]	Lack of login bruteforce protection	Low	Fixed
[L-05]	Access and refresh JWT share the same secret	Low	Fixed

ID	Title	Severity	Status
[L-06]	Account enumeration and email spam via forgot-password	Low	Fixed
[L-07]	Sensitive token exposure in JSON responses	Low	Fixed
[I-01]	Symmetric JWT Signing	Informational	Fixed

Detailed Findings

[H-01] Privilege Escalation - wallet verified flag

Severity

Impact:

High

Likelihood:

High

Description

The `UpdateWalletAccountDto` includes an optional boolean field named `verified`. This allows any authenticated user to send a `PATCH` request with `{\"verified\": true}` to the `/wallets/accounts/{walletId}` endpoint and arbitrarily mark their own wallet as verified. This completely bypasses the intended cryptographic verification flow, which should be the only way to achieve a verified status.

Recommendations

We recommend that you immediately remove the `verified` field from the `UpdateWalletAccountDto`. Client side applications should never be allowed to directly set a security sensitive status like this. The `verified` flag should be managed exclusively by backend logic and only be set to `true` after a successful and complete cryptographic ownership verification process (e.g. after a successful signature check).

[H-02] State changing GET and sensitive token exposure in body

Severity

Impact:

High

Likelihood:

High

Description

The `GET /auth/session/privy-token` endpoint has multiple critical security flaws. First, it uses the HTTP GET method to perform a state changing action (generating and setting a new token), which violates HTTP semantics and makes it vulnerable to CSRF attacks that do not require a preflight request. Second, and more critically, it returns the sensitive session token directly in the JSON response body while also setting it as an `HttpOnly` cookie. This completely defeats the security benefit of the `HttpOnly` flag, as the token can be easily stolen via a Cross Site Scripting (XSS) attack, even though the cookie itself is protected from script access.

Recommendations

We recommend that you immediately refactor this endpoint. It should be changed to use the HTTP POST method, and it must be protected by the application's CSRF defense mechanism. Most importantly, the token value must be removed from the JSON response body. The endpoint should only set the token in the `HttpOnly` cookie and return a success status. Additionally, we recommend adding `Cache-Control: no-store` and `Pragma: no-cache` headers to all responses that set or modify authentication tokens to prevent them from being cached by browsers or intermediate proxies.

[H-03] CSRF risk due to cookie based JWT with SameSite=None

Severity

Impact:

High

Likelihood:

High

Description

The application uses cookie based authentication where JWTs are stored in `HttpOnly` cookies. In production, these cookies are configured with `SameSite=None`, which means browsers will send them with cross origin requests. While the application implements CSRF protection (`XhrCsrfMiddleware`), it is only applied to routes under the `/wallets/*` path. This leaves all other state changing endpoints, such as `/auth/change-password`, `/auth/profile`, and other user account management routes, completely unprotected from Cross-Site Request Forgery (CSRF) attacks. An attacker can host a malicious website that tricks a logged in user's browser into sending authenticated requests to these unprotected endpoints, allowing the attacker to perform unauthorized actions on the victim's behalf.

Recommendations

We recommend that you immediately expand the scope of the `XhrCsrfMiddleware` to protect all routes by changing the configuration to `forRoutes(\'*\])`. This will ensure that all state changing endpoints are protected from CSRF attacks. For a more robust long term solution, we recommend setting `SameSite=Lax` as the default for all authentication cookies to provide a strong first layer of defense against CSRF. You should also consider moving away from cookie based authentication for API endpoints and prefer `Authorization: Bearer` tokens, which are not vulnerable to CSRF.

[H-04] NoSQL/MongoDB operator injection in update endpoints

Severity

Impact:

High

Likelihood:

High

Description

The `updateWallet` and `updateWalletByAddress` methods in the `WalletAccountsService` are vulnerable to NoSQL injection. The `update` object, which is derived directly from the request body, is passed into the `findOneAndUpdate` Mongoose method without any sanitization or validation to prevent MongoDB operators. An authenticated attacker can craft a request body containing keys like `$set`, `$unset`, or `$rename` to modify arbitrary fields in their own wallet document in the database. This could allow them to corrupt data, bypass business logic, or escalate privileges (e.g. by setting `verified: true`).

Recommendations

We recommend enabling the `whitelist: true` and `forbidNonWhitelisted: true` options in the global `ValidationPipe` in `main.ts`. This will strip out any properties from the request body that are not explicitly defined in the DTO, including MongoDB operators. For a more robust, long term solution, we recommend that you never pass user controlled objects directly to database update operations. Instead, you should create a new object on the server side containing only the fields that are allowed to be updated and wrap it in a `$set` operator.

[H-05] Second-order denial of service

Severity

Impact:

High

Likelihood:

Medium

Description

In `apps/backend/src/swap/swap.service.ts`, the `getWalletAssets` function acts as an unrestricted proxy to third-party blockchain data providers. It accepts an arbitrary address and immediately triggers external API calls to Solscan or DeBank. Because there is no verification that the address belongs to the authenticated user, an attacker can loop through random addresses to exhaust the platform's API credits or rate limits with these providers. This is a second-order denial of service as it will occur between the API and backend services (Solscan, Debank) effectively blocking it for the whole API.

The calls to the external services are visible in the conditional block below:

```
// apps/backend/src/swap/swap.service.ts

async getWalletAssets(address: string):
Promise<GetWalletAssetsResponseDto> {
  // ... validation logic ...

  if (isSolana) {
    // 1. Direct call to Solscan Service with arbitrary address
    assets = await this.solscanService.getAssetFromWallet(address)
  } else if (isEvm) {
    // 2. Direct call to DeBank Service with arbitrary address
    assets = await this.debankService.getAssetFromUserAllTokenList({
      id: address,
      is_all: false,
    })
  }

  // ...
}
```

Recommendations

Validate that the address exists in the authenticated user's linked wallets (e.g., via WalletAccountsService) before making the external API call. Consider adjusting the rate limit so the total front API rate limit is more strict than rate limit on backend to external apis.

[H-06] Broken Access Control allowing Waitlist Bypass

Severity

Impact:

High

Likelihood:

High

Description

In `apps/backend/src/auth/auth.controller.ts`, the application implements a waitlist mechanism to gate user access (user.onWaitlist). However, the endpoint responsible for skipping this waitlist is misconfigured to allow access to any authenticated user, regardless of their privileges. The `skipWaitlist` endpoint includes `UserRole.USER` in its allowed roles. This allows any newly registered user (who is on the waitlist) to immediately call this endpoint and activate their own account, rendering the waitlist logic ineffective.

```
@Post('skip-waitlist')
@Roles(UserRole.USER, UserRole.ADMIN) // <--- Vulnerability: Allows
standard users
async skipWaitlist(@ActiveUser('sub') userId: string) {
  await this.authService.skipWaitlist(userId)
  return {
    success: true,
  }
}
```

Recommendations

Restrict access to this endpoint to administrators only, or remove the endpoint if it was intended solely for development testing.

[M-01] Race Condition - wallet subscription limit

Severity

Impact:

Medium

Likelihood:

Medium

Description

The `addOnchainWallet` method suffers from a classic TOCTOU (Time-Of-Check-Time-Of-Use) race condition. It first checks the user's current wallet count against their subscription limit, and then, in a separate, non atomic operation, it acts by inserting the new wallet. An attacker can exploit this by sending multiple concurrent requests to create wallets. Several requests can pass the initial check before the database is updated, allowing the user to create more wallets than their subscription plan allows.

Recommendations

We recommend that you refactor this logic to be atomic. The ideal solution is to use a database transaction or an atomic update operation. For example, you could maintain a separate usage counter document for each user and use `findOneAndUpdate` with a condition like `currentCount < limit` while atomically incrementing the count. If transactions are not feasible, a distributed lock (e.g., using Redis) could be used to ensure that only one wallet creation request per user is processed at a time. As an immediate, temporary mitigation, we recommend adding monitoring to detect and alert on rapid wallet additions from a single user.

[M-02] Unbounded credentials length enables DoS

Severity

Impact:

Medium

Likelihood:

Medium

Description

The `apiKey`, `apiSecret`, and `passphrase` fields in the `AddExchangeAccountDto` do not have any `MaxLength` validation constraints. This allows an authenticated attacker to send requests with extremely large values for these fields (up to the body parser limit of 100KB). The server then performs expensive encryption operations on these large payloads and stores them in the database, consuming significant CPU, memory, and storage resources. While the risk is partially mitigated by a global rate limit, it still presents a vector for denial of service.

Recommendations

We recommend that you add `@MaxLength` constraints to all sensitive string fields, especially those that are encrypted or stored in the database. A reasonable limit, such as 512 characters, should be sufficient for API keys and secrets. This will ensure that oversized payloads are rejected early in the validation stage, before any expensive processing occurs.

[M-03] CSRF and abuse on public refresh-token rotation

Severity

Impact:

Medium

Likelihood:

Medium

Description

The `POST /auth/refresh-token` endpoint is a public endpoint that relies on the `refresh_token` cookie for authentication. It lacks both CSRF protection and rate limiting. This allows a malicious website to trigger a cross-site request to this endpoint, causing the victim's refresh token to be rotated. While the attacker cannot steal the new tokens, they can cause repeated, unwanted token rotations, which could lead to session disruption or denial of service for the victim. It also creates unnecessary load on the server.

Recommendations

We recommend that you add CSRF protection to this endpoint, as it is a state changing action that relies on cookies. Since this is a public endpoint (not requiring prior authentication), you should implement a custom header check mechanism where the client must include a specific header like X-Requested-With: XMLHttpRequest or X-App-Request: true in the request. Alternatively, you can implement a double submit cookie pattern where a CSRF token is set in a readable cookie and must be echoed back in a custom request header. The server should verify that both values match before processing the request.

[M-04] Swap history manipulation

Severity

Impact:

Medium

Likelihood:

Medium

Description

In `apps/backend/src/swap/swap.controller.ts` at handler `@Post('history/init')`, and further in `apps/backend/src/swap/swap.service.ts`, user can arbitrarily create swap history. Calling the endpoint inserts the record into database but there is no on-chain verification if such transaction ever took place. The `initSwapHistory` method allows the caller to define the initial status of a swap transaction via the DTO. An attacker can call this endpoint with status: 'DONE' to inject fake completed swaps.

```
  async initSwapHistory(userId: string, dto: PostSwapHistoryInitDto):
Promise<SwapHistory> {
    try {
      const created = await this.swapHistoryModel.create({
        userId,
        fromChain: dto.fromChain,
        toChain: dto.toChain,
        fromToken: dto.fromToken,
        toToken: dto.toToken,
        fromAmount: dto.fromAmount,
        toAmountEstimated: dto.toAmountEstimated,
        toTokenPriceAtSwap: dto.toTokenPriceAtSwap,
        slippage: dto.slippage,
        routeSummary: dto.routeSummary,
        steps: dto.steps ?? [],
        txHashes: dto.txHashes ?? [],
        status: dto.status ?? SwapHistoryStatus.INITIATED,
        fromAddress: dto.fromAddress,
        toAddress: dto.toAddress,
        type: dto.type ?? 'swap',
      })
      return created.toObject() as unknown as SwapHistory
    } catch (error) {
      this.logger.error('Failed to init swap history', error)
      throw new BadRequestException('Failed to init swap history')
    }
  }
```

Recommendations

Hardcode the initial status to INITIATED in the service method. Transitions to DONE or FAILED should only occur via internal polling logic or verified webhooks/callbacks. To be 100% sure about the validity of transaction, an on-chain hash and transaction parsing should exist.

[M-05] Authentication Bypass of 2FA via Mobile Platform Flag

Severity

Impact:

High

Likelihood:

Low

Description

In `apps/backend/src/auth/services/two-factor-check.service.ts`, the 2FA does not work for `mobile` platform by design. The application explicitly bypasses Two-Factor Authentication (2FA) checks when the platform parameter is set to 'mobile'. The public endpoint `POST /auth/mobile/sign-in` sets this flag automatically. An attacker with compromised credentials (email/password) can bypass 2FA protection simply by sending their login request to the mobile endpoint instead of the web endpoint.

```
if (platform === "mobile") {
  this.logger.log(`2FA skipped for mobile platform, user ${user._id}`);
```

```
    return { requires2FA: false, user: userWith2FA };
  }
```

Recommendations

Remove the bypass for mobile platforms. Enforce 2FA consistently across all clients. If the mobile app uses a different auth mechanism (e.g., biometrics), validation should occur via a secure token exchange or signed device attestation, not a simple string flag in the API call.

[M-06] Missing Rate Limiting allows Referral Code Bruteforcing

Severity

Impact:

Medium

Likelihood:

Medium

Description

The `submitReferralCode` endpoint allows users on the waitlist to submit a code to gain immediate access to the platform. Unlike other inputs in the controller, this endpoint lacks the `@Throttle` decorator.

Because there is no limit on the number of attempts a single user/IP can make, a malicious user can automate thousands of requests against the `handleSubmitReferralCode` service method to guess valid referral codes.

In `apps/backend/src/auth/auth.controller.ts`:

```
@Post('submit-referral-code')
@Roles(UserRole.USER, UserRole.ADMIN)
async submitReferralCode(...)
```

In `apps/backend/src/users/referral.service.ts`:

```
async handleSubmitReferralCode(userId: string, referralCode: string):
Promise<void> {
  // ... checks user status ...

  // Database lookup occurs on every request
  const referrerId = await this.validateReferralCode(referralCode)

  if (!referrerId) {
    throw new BadRequestException('Invalid referral code') // Feedback
loop for bruteforce
  }
  // ...
}
```

Recommendations

Apply the existing throttling mechanism to this endpoint.

[L-01] Precision Loss in Token Amount Parsing

Description

In `apps/web/components/swap/utils/amount.ts`, in functions `parseTokenAmount` and `toWei`, decimals number is casted to a `BigInt`:

```
const divisor = BigInt(10 ** decimals);
```

The calculation `10 ** decimals` uses JavaScript numbers. For tokens with high decimal precision (e.g., >15), this can lead to precision loss or overflow before conversion to BigInt.

Recommendations

Use BigInt arithmetic entirely: `BigInt(10) ** BigInt(decimals)`.

[L-02] Raw Error Message Exposure

Description

Unmatched error patterns return raw provider messages to users, potentially leaking implementation details. While most of the errors are handled, the most dangerous in terms of information leakage - unhandled ones - are thrown to users in their raw form. Unexpected errors may in worst cases include source code, environment variables or other secrets. This is used commonly in the application, e.g. in whole `apps/web/lib/swap/*` files.

In `apps/web/lib/swap/execute-transaction.ts`:

```

  // Return original message if no specific match
  return new Error(error.message)
```

Recommendations

For the unexpected errors, log the error details server side, but print just "unexpected error" to users.

[L-03] Missing HTTP Security Headers

Description

The provided source code (Controllers and Services) does not show any implementation of standard HTTP security headers. While cookies are configured with Secure and HttpOnly flags, the response headers that protect the browser context itself appear to be missing. The related issues are:

- Clickjacking and XSS: Content-Security-Policy: frame-ancestors 'none', and a configuration of CSP against external scripts, an attacker can load the application inside an `<iframe>` on a malicious site. They can then overlay invisible elements to trick users into clicking sensitive buttons (e.g., "Connect Wallet" or "Sign Transaction"). Additionally, any occurrence of HTML injections can be turned into an XSS (unless the frontend layer sanitizes the content - but with modern react it usually does)
- MIME-Type Sniffing: Without X-Content-Type-Options: nosniff, browsers may interpret files as a different MIME type than declared, potentially leading to XSS if users can upload content (e.g., avatars).
- Protocol Downgrade: Without Strict-Transport-Security (HSTS), users are vulnerable to SSL stripping attacks, which is critical here because the application uses SameSite: 'none' cookies (which require a secure context).

Recommendations

Install and configure Helmet middleware in the application, then use it to implement mentioned security headers

[L-04] Lack of login bruteforce protection

Description

In `apps/backend/src/auth/auth.controller.ts` and `apps/backend/src/auth/auth.service.ts`, the application protects authentication endpoints (sign-in, sign-up) using the `@Throttle` decorator, which typically enforces rate limits based on the requestor's IP address (10 requests per minute).

However, there is no logic in the AuthService.validateUser method to track failed login attempts or enforce an account-level lockout. This creates a vulnerability to Distributed Brute Force attacks. An attacker can utilize a botnet (rotating IP addresses) to guess passwords against a specific target account. Since the rate limit is per-IP, the target account never locks, allowing the attacker unlimited guesses over time.

```
@Post('sign-in')
@Throttle({ default: { limit: 10, ttl: 60000 } }) // Limits by IP
async signIn(...)
```

Recommendations

Implement an account-level lockout mechanism:

- Add `failedLoginAttempts` and `lockoutUntil` fields to the User model.
- If attempts exceed a threshold (e.g., 5), set `lockoutUntil` to 15 minutes in the future and require password change e.g. forgot password like for that email.

[L-05] Access and refresh JWT share the same secret

Description

The application uses the same JWT secret for signing both short lived access tokens and long lived refresh tokens. While not a direct vulnerability, this practice is not recommended as it increases the impact if the secret is ever compromised. A compromised secret would allow an attacker to forge both access and refresh tokens. It can also lead to token confusion attacks where a token intended for one purpose (e.g., access) could potentially be accepted for another (e.g. refresh), although this is unlikely with the current implementation.

Recommendations

We recommend that you use a distinct, separate secret for signing refresh tokens. This can be configured via a new environment variable (e.g., `JWT_REFRESH_SECRET`). As a further defense in depth measure, we recommend enforcing the `aud` (audience) and `type` claims during token verification to ensure that a token is only used for its intended purpose. For example, an access token should have `type: 'access'` and a refresh token should have `type: 'refresh'`.

[L-06] Account enumeration and email spam via forgot-password

Description

The `POST /auth/forget-password` endpoint exhibits different behavior depending on whether the provided email address exists in the database. It returns a specific error message if the user is not found, which allows an attacker to enumerate registered email addresses.

Recommendations

We recommend that you modify the endpoint to return a generic success message in all cases, regardless of whether the email address was found in the database. This prevents an attacker from being able to distinguish between registered and unregistered accounts. Additionally, we recommend applying a strict rate limit to this endpoint (e.g., 5 requests per minute per IP address) to prevent email spamming and brute-force attacks. For further protection, you could consider adding a CAPTCHA challenge after a certain number of failed attempts.

[L-07] Sensitive token exposure in JSON responses

Description

Several authentication related endpoints, including `signIn`, `signUp`, and `refreshToken`, return access and refresh tokens directly in the JSON response body. This practice increases the attack surface for token theft. If an XSS vulnerability were to be discovered elsewhere in the application, an attacker could easily steal these tokens from the JSON response. This is especially risky when `HttpOnly` cookies are also being used, as it negates their primary security benefit.

Recommendations

We recommend that for all web based authentication flows, you remove the tokens from the JSON response body and rely exclusively on `HttpOnly` cookies to store and transmit them. For mobile flows that require tokens in the response, we recommend ensuring that the access tokens are short-lived. Additionally, we recommend adding `Cache-Control: no-store` headers to all responses that contain tokens to prevent them from being stored in browser caches or by intermediary proxies.

[I-01] Symmetric JWT Signing

Description

In `apps/backend/src/auth/auth.service.ts`, the application generates sensitive short-lived tokens (Password Reset and Email Verification) using symmetric secrets (HMAC) via jwtService.signAsync. Currently, the code does not explicitly specify the signing algorithm in the configuration options, relying instead on the library's default behavior (likely HS256 given a secret is provided).

If an attacker obtains a single valid JWT (e.g., from their own email verification link), they can attempt to guess the secret on their own high-speed hardware (Offline). If the secret in .env file is short or simple (e.g., secret123), it can be cracked. With symmetric keys, verification requires the secret in the same way as public/private keys, making it impossible to crack the token.

Recommendations

Explicitly set { algorithm: 'HS256' } in the signing options to prevent potential algorithm confusion attacks. Consider migrating these flows to use Asymmetric Keys (RS256).