Security Audit Report

Dither Service

v1.0

September 23, 2025

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by All in Bits, Inc. to perform a security audit of Dither Service Security Audit and Penetration Test.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/allinbits/dither-service
Commit	`bc440f514ad69035a7c4108da7e2d65c94e517b8`
Scope	Line-by-line security audit of the following components: ● `https://github.com/allinbits/dither-service/tree/main/packages/api-main` ● `https://github.com/allinbits/dither-service/tree/main/packages/lib-api-types` ● `https://github.com/allinbits/dither-service/tree/main/packages/reader-main` `https://github.com/allinbits/dither-service/blob/main/packages/nginx/default.conf` Penetration testing of a running instance of the frontend component: ● `https://github.com/allinbits/dither-service/tree/main/packages/frontend-main`
Fixes verified at commit	`8919c72eae4e16327a8d76bcc9771b73f5d71ed9`

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Dither Service is a provider engine for individual front-end applications to listen to the Dither Protocol. The Dither Service uses event-sourcing data to rebuild application state from Cosmos based chains, specifically AtomOne.

The service provides functionalities based on two primary roles:

- Users: Can perform a range of social actions, including creating posts, replying to others, following and unfollowing users, and interacting with content through likes, dislikes, and flags.
- Moderators: A privileged role with the ability to manage the community by removing or restoring posts, and banning or unbanning users.

The application uses hybrid cookie/JWT based authentication. These tokens are managed through secure, httpOnly cookies after a user successfully signs in with their crypto wallet. All the data, user interactions, and moderation logs are stored in a PostgreSQL database.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**Medium**	-
Code readability and clarity	**Medium-High**	-
Level of documentation	**Medium**	-
Test coverage	**N/A**	-

Summary of Findings

No	Description	Severity	Status
1	Weak default reader `AUTH` shared secret	Major	Resolved
2	Weak default JWT secret allows token forgery	Major	Resolved
3	Unsafe moderation logic can restore unrelated content	Major	Resolved
4	Client-side logout fails for `httpOnly` cookie	Major	Resolved
5	JWT verification does not restrict algorithms	Minor	Resolved
6	Insecure protocol memo construction may lead to injection attacks	Minor	Resolved
7	Verbose error disclosure in API responses	Minor	Resolved
8	Unsafe cookie attributes	Minor	Resolved
9	Duplicate route registrations	Minor	Resolved
10	State-changing action exposed as GET	Minor	Resolved
11	No rate limiting or abuse protection on authentication endpoints	Minor	Resolved
12	Oversized content not validated on reply endpoint	Minor	Acknowledged
13	Potential SQL injection via interpolated table name	Informational	Acknowledged
14	Potential timing attack in authorization check	Informational	Resolved
15	Unused code in the frontend	Informational	Resolved

Detailed Findings

1. Weak default reader `AUTH` shared secret

Severity: Major

The writer API trusts a shared `Authorization` header that is compared to a server side secret `AUTH` in `packages/api-main/src/config.ts:45-63` and because the configuration defaults `AUTH` to default an attacker can send `Authorization: default` to call writer endpoints such as /v1/post, `/v1/update-state` and other write endpoint.

Recommendation

We recommend aborting startup whenever `AUTH` is unset or equals a known placeholder using a long random secret stored securely rotating it when appropriate and enforcing the check in middleware for every writer route for example if `(!process.env.AUTH || process.env.AUTH === 'default') { throw new Error('AUTH must be set to a strong secret') }` and consider replacing the shared header with signed requests or mutual TLS between the reader and the API while returning 401 on missing or mismatched Authorization headers.

Status: Resolved

2. Weak default JWT secret allows token forgery

Severity: Major

The server falls back to a hardcoded JWT secret default secret-key when the JWT environment variable is absent in `packages/api-main/src/config.ts:45-63` which allows an attacker to forge `HS256` tokens and impersonate arbitrary users leading to actions such as reading or marking another user's notifications.

Recommendation

We recommend requiring a strong unpredictable JWT secret and aborting startup if JWT is unset or equals the default value.

Status: Resolved

3. Unsafe moderation logic can restore unrelated content

Severity: Major

In `packages/api-main/src/posts/mod.ts:209`, the `ModUnban` function restores all posts by an author that were previously removed by any moderator.

This logic is too broad and can lead to unintended data restoration. For instance, if a user's post was removed for a specific reason. such as spam, and they were later banned and then unbanned for a separate incident, the original spam post would be incorrectly restored.

It was also noted that deleting comments in fact does not delete them from the database, but marks as removed while maintaining their content.

Recommendation

We recommend changing the moderation logic to be more granular. Each moderation action like a post removal or ban should be a distinct event. Removal of a comment should delete it from the database.

An "unban" action should only revert the post removals associated with that specific ban event, not all removals ever performed by a moderator on that user.

Status: Resolved

4. Client-side logout fails for `httpOnly` cookie

Severity: Major

In `packages/frontend-main/src/composables/useWallet.ts:81`, the logout functionality on the frontend attempts to clear the authentication cookie via `document.cookie`.

This fails because the server correctly sets the cookie with the `httpOnly` flag, which prevents it from being accessed or modified by client-side JavaScript. As a result, the logout feature is broken, and users remain logged in.

Recommendation

Create a dedicated server-side logout endpoint (e.g., `/logout`). The client should send a request to this endpoint to properly invalidate the session on the server side.

Status: Resolved

5. JWT verification does not restrict algorithms

Severity: Minor

The application verifies JSON Web Tokens without restricting acceptable algorithms in `packages/api-main/src/shared/jwt.ts:10-31` The current `verifyJWT` call uses `jsonwebtoken.verify(token, JWT)` without an explicit allow list which increases the risk of algorithm confusion if token headers are manipulated even though the library rejects the none algorithm.

Recommendation

We recommend explicitly constraining both signing and verification to the expected algorithm and setting an appropriate expiration.

Status: Resolved

6. Insecure protocol memo construction may lead to injection attacks

Severity: Minor

The frontend constructs protocol memos with unescaped user input in `packages/frontend-main/src/composables/useCreatePost.ts:22-72`

For example `dither.Post("${message}")` embeds a message directly so a double quote inside a message can terminate the string, break the memo format and confuse the on chain parser which may cause malformed posts replies or other unexpected behavior.

Recommendation

We recommend escaping or encoding memo arguments before interpolation.

Status: Resolved

7. Verbose error disclosure in API responses

Severity: Minor

The API responds with verbose error messages in `packages/api-main/src/posts/mod.ts` at lines `58-63`, `120-126`, `177-182`, and `231-236` returning raw error strings like `'failed to delete post, maybe invalid: ' + err` which can expose stack traces or internal details to clients and aid reconnaissance or targeted abuse.

Recommendation

We recommend logging detailed errors only on the server using `console.error` or a centralized logger and returning uniform generic responses with appropriate status codes and enforcing this via a global error handler or middleware that maps exceptions to safe client messages.

Status: Resolved

8. Unsafe cookie attributes

Severity: Minor

The session authentication cookie is issued without the `Secure` attribute in `packages/api-main/src/posts/auth.ts:20-38` and although `httpOnly` and `sameSite` are set the absence of secure true can allow the cookie to transmit over plaintext connections if any HTTP path exists and modern browsers will reject `SameSite=None` without `Secure` which together with `JWT_STRICTNESS` permitting none increases CSRF and session exposure risks.

Recommendation

We recommend always setting the `Secure` attribute to `True` in production and defaulting `SameSite` to `Lax` or `Strict`.

Status: Resolved

9. Duplicate route registrations

Severity: Minor

The server registers the same routes twice in `packages/api-main/src/index.ts:31-64` specifically `app.get('/last-block', ...)` and `app.get('/notification-read', ...)` are duplicated which makes routing fragile and order dependent and can lead to confusing maintenance.

Recommendation

We recommend removing the duplicate registrations so each path is mounted exactly once, consolidating middleware and handlers for those routes.

Status: Resolved

10. State-changing action exposed as GET

Severity: Minor

The frontend triggers a state change via `/notification-read` in `packages/frontend-main/src/composables/useReadNotification.ts:24-47` which violates HTTP best practises and can be unintentionally replayed by caches or link prefetchers and increases CSRF exposure since browsers automatically include cookies on GET. This allows for a limited CSRF attack with low impact such as making a user unintentionally have their notifications marked as read.

Recommendation

We recommend setting the server to accept only POST for this endpoint.

Status: Resolved

11. No rate limiting or abuse protection on authentication endpoints

Severity: Minor

The authentication endpoints lack rate limiting in `packages/api-main/src/index.ts:66-85` where unauthenticated routes `/v1/auth-create` and `/v1/auth` accept unbounded traffic that can be abused to flood the service inflate AuthRequests rows and force excessive signature verification leading to database and CPU exhaustion and denial of service.

Recommendation

We recommend enforcing IP and user scoped rate limits with request throttling on both endpoints returning `429` on excess using short TTL buckets or token buckets deploying CAPTCHA.

Status: Resolved

12. Oversized content not validated on reply endpoint

Severity: Minor

In `packages/api-main/src/index.ts`, the `/reply` endpoint does not validate the length of the message content. While the database schema enforces a 512-character limit, failing to validate input at the application layer can lead to unhandled database errors and inconsistent API behavior.

Recommendation

Add an application-level validation check in the `/reply` handler to ensure the message length does not exceed the 512-character limit before attempting to save it to the database.

Status: Acknowledged

13. Potential SQL injection via interpolated table name

Severity: Informational

The utility function `getJsonbArrayCount(hash, tableName)` interpolates the tableName into a raw SQL string in `packages/api-main/src/utility/index.ts:44-55` and although current calls use static names this pattern creates a latent SQL injection risk because identifiers cannot be safely parameterized and a user influenced tableName could escape the query and execute unintended SQL

Recommendation

We recommend eliminating the generic usage or constraining `tableName` to a strict compile time enum of allowed tables using Drizzle identifier helpers.

Status: Acknowledged

14. Potential timing attack in authorization check

Severity: Informational

The reader authorization key in `packages/api-main/src/utility/index.ts:47` is validated using the standard string comparison operator (===). This operator terminates as soon as a character mismatch is found, which can theoretically allow an attacker to discern the secret key character-by-character by measuring response times. While the practical risk is very low, it is not best practice for comparing secrets.

Recommendation

We recommend using a constant-time comparison function, such as `crypto.timingSafeEqual` to validate the authorization header. This ensures the comparison always takes the same amount of time, mitigating any potential for timing attacks.

Status: Resolved

15. Unused code in the frontend

Severity: Informational

In `packages/frontend-main/src/utility/sanitize.ts:5`, the `purifyHtml` function, which appears to be intended for sanitizing HTML to prevent Cross-Site Scripting attacks is defined but is not used anywhere in the frontend codebase. Unused code adds to maintenance overhead and can cause confusion.

Recommendation

Remove the `purifyHtml` function and its related imports or use it within the codebase.

Status: Resolved