Security Audit Report

ZIGChain Reward Contract

v1.0

November 11, 2025

Table of Contents

Table of Contents 2
License 4
Disclaimer 5
Introduction 6
Purpose of This Report 6
Codebase Submitted for the Audit 6
Methodology 7
Functionality Overview 7
How to Read This Report 8
Code Quality Criteria 9
Summary of Findings 10
Detailed Findings 12
1. Users can claim past their allowed_amount via cumulative cap bypass 12
2. Incorrect decimal calculations in token amount validations 12
3. setRewards lacks anti-spam mechanism 13
4. Incorrect address validation in setRewards allows registration to arbitrary EVM wallets 13
5. Missing contract address validation in claim event parsing allows spoofed database entries 14
6. Missing ValidationPipe and class-transformer enables unvalidated input and injection attacks 14
7. Insufficient validation of reward token denomination 15
8. Lack of API rate limiting 15
9. Missing validation in admin change allows setting invalid or inaccessible admin addresses 16
10. Case-insensitive address check allows duplicate reward registrations 16
11. Token denomination updates invalidate existing claim records 16
12. Missing validation of ZigWallet signer enables reward registration to arbitrary addresses 17
13. Missing rate limiting creates documentation inconsistency 17
14. Missing aggregate allocation tracking in set_user_reward could lead to contract insolvency 18
15. Missing pagination and result limits on database queries causes unbounded resource consumption 18
16. Lack of input validation in multiple instances 19
17. parseMessage regex should implement max message length validation 20
18. Remove unused code blocks 20
19. Missing check to prevent pausing an already paused contract 20
20. Hardcoded gas price prevents flexible fee adjustment and creates operational risk 21

21. Unused getClaimsByWalletAddress function 21
22. Unregistered IpLoggerMiddleware 21
23. Typo in CORS configuration 22

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 Highend Technologies LLC to perform a security audit of the ZIGChain rewards contract and API.

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/ZIGChain/staking-rewards-private/
Commit	`6f45da4c1550e1b101fb751c30c629729db6637a`
Scope	`/zig-claim-api` `/zig-claim`
Fixes verified at commit	`a5c9023b088700a69e863bbad403af3d61167aee` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes, such as additional features, have not been reviewed.

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

The ZIGChain reward system consists of a CosmWasm smart contract that manages airdrop rewards with configurable user allocations, rate-limited claims, automated penalty calculations, and administrative controls for pausing operations and withdrawing unclaimed tokens, paired with a NestJS RESTful API that provides endpoints for managing rewards, wallet information, and user rankings on the ZIGChain blockchain.

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-High**	-
Code readability and clarity	**Medium-High**	-
Level of documentation	**High**	-
Test coverage	**Medium**	- `zig-claim` reported 81.66% coverage (`cargo tarpaulin`) - No test coverage for `zig-claim-api`

Summary of Findings

No	Description	Severity	Status
1	Users can claim past their `allowed_amount` via cumulative cap bypass	Critical	Resolved
2	Incorrect decimal calculations in token amount validations	Critical	Resolved
3	`setRewards` lacks anti-spam mechanism	Major	Resolved
4	Incorrect address validation in `setRewards` allows registration to arbitrary EVM wallets	Major	Resolved
5	Missing contract address validation in claim event parsing allows spoofed database entries	Major	Resolved
6	Missing ValidationPipe and class-transformer enables unvalidated input and injection attacks	Major	Resolved
7	Insufficient validation of reward token denomination	Minor	Resolved
8	Lack of API rate limiting	Minor	Resolved
9	Missing validation in admin change allows setting invalid or inaccessible admin addresses	Minor	Resolved
10	Case-insensitive address check allows duplicate reward registrations	Minor	Resolved
11	Token denomination updates invalidate existing claim records	Minor	Resolved
12	Missing validation of `ZigWallet` signer enables reward registration to arbitrary addresses	Minor	Resolved
13	Missing rate limiting creates documentation inconsistency	Minor	Resolved
14	Missing aggregate allocation tracking in `set_user_reward` could lead to contract insolvency	Minor	Acknowledged
15	Missing pagination and result limits on database queries causes unbounded resource consumption	Minor	Acknowledged
16	Lack of input validation in multiple instances	Minor	Resolved

17	`parseMessage` regex should implement max message length validation	Informational	Resolved
18	Remove unused code blocks	Informational	Resolved
19	Missing check to prevent pausing an already paused contract	Informational	Resolved
20	Hardcoded gas price prevents flexible fee adjustment and creates operational risk	Informational	Resolved
21	Unused `getClaimsByWalletAddress` function	Informational	Resolved
22	Unregistered `IpLoggerMiddleware`	Informational	Resolved
23	Typo in CORS configuration	Informational	Resolved

Detailed Findings

1. Users can claim past their `allowed_amount` via cumulative cap bypass

Severity: Critical

The `claim` function in `zig-claim/src/contract/exec.rs:302`, the `claim` function computes claims based on daily amounts with rewards applied, but lacks validation to ensure that cumulative claims (`info.claimed_amount + reward`) do not exceed `allowed_amount`. This enables users to claim approximately 115% of their allocation - for example, a user with 180,000,000 token allocation can extract up to 208,000,000 tokens by continuing to claim after the 180-day period.

Additionally, there is no enforcement of the `REWARD_PERIOD_DAYS` limit, allowing claims to continue indefinitely as long as the contract maintains a balance. The core issue is the absence of a cumulative validation check rather than just the time period extension.

Recommendation

We recommend adding validation to ensure that `info.claimed_amount + reward <= allowed_amount` before processing any claim, and enforcing the reward period limit. Note that relying on owner withdrawal as a mitigation would harm legitimate users who have not yet claimed their allocations.

Status: Resolved

2. Incorrect decimal calculations in token amount validations

Severity: Critical

In `zig-claim/src/contract/exec.rs:94`, `507`, and multiple instances in `zig-claim/src/tests.rs`, the contract contains decimal calculation errors where comments claim values represent billions of tokens with 18 decimals, but actual values are 9 orders of magnitude smaller:

- `MAX_ALLOWED_AMOUNT`: 1 token (instead of 1 billion)
- `MAX_FUND_AMOUNT`: 10 tokens (instead of 10 billion)

This results in extremely restrictive validation limits that would prevent normal contract operation.

Recommendation

We recommend fixing the constants `MAX_FUND_AMOUNT` and `MAX_ALLOWED_AMOUNT` to match their comments. Additionally, update all test values to use correct decimal calculations or adjust comments to reflect actual values.

Status: Resolved

3. `setRewards` lacks anti-spam mechanism

Severity: Major

In the `setRewards` function in `zig-claim-api/src/core/claim/claim.service.ts:30-63`, the code performs signature verification and queries rewards data but immediately broadcasts a blockchain transaction in `line 60` without first checking if the user is already registered. While the smart contract correctly rejects duplicate registrations, the request still consumes admin gas for broadcasting each duplicate attempt. An attacker with a whitelisted address can repeatedly call this endpoint with valid signatures; each call drains admin wallet gas even though the contract rejects it. By making hundreds or thousands of rapid calls, an attacker can exhaust all gas tokens from the admin wallet, causing a DoS that blocks legitimate operations as well as having an economic impact on the protocols funds allocated to gas payments.

Recommendation

We recommend querying the contract to verify the user is not already registered before broadcasting. If already set, return an error before sending the tx to avoid wasting gas. Alternatively, add rate limiting tied to the EVM wallet address.

Status: Resolved

4. Incorrect address validation in `setRewards` allows registration to arbitrary EVM wallets

Severity: Major

In the `setRewards` function in `zig-claim-api/src/core/claim/claim.service.ts:44`, the code uses a conditional `(isAddress(evmWallet) != isAddress(signer))` which checks if both addresses are valid format (returns boolean), but never verifies that the evmWallet extracted from the message matches the signer who cryptographically signed it.

An malicious user who controls a whitelisted EVM address and their own ZIGChain wallet can sign a message containing a different EVM address, and the system will register the reward to the address in the message while verifying the signature from the attacker's address. This allows a malicious user to "reserve" other users' eligible EVM addresses by registering them to arbitrary ZIGChain wallets, effectively blocking legitimate users from claiming their rewards. While this does not directly steal funds (the victim's reward still exists in the contract), it prevents legitimate users from accessing their allocations. An example could be if a user is disgruntled by their small allocation and chooses to query the leaderboard and target users with the highest allocation. This could cause a large economic impact for those users.

Recommendation

We recommend updating the function to return an error if `evmWallet.toLowerCase() !== signer.toLowerCase()`.

Status: Resolved

5. Missing contract address validation in claim event parsing allows spoofed database entries

Severity: Major

In the `extractClaimAttributes` function in `zig-claim-api/src/core/claim/claim.service.ts:104-122`, the code extracts event attributes without verifying which contract emitted the event. Line `106` searches for any wasm event type without checking the contract address. An attacker could deploy a malicious CosmWasm contract that emits events with matching attribute keys but arbitrary values, or use a transaction hash from another legitimate contract that happens to have a wasm event with similar attributes.

When the attacker provides this transaction hash to the claim endpoint in lines `72-102`, the system will accept the event and insert fake claim data into the database line `90`, including any wallet address and amount specified in the spoofed event.

While this does not directly move funds on-chain, it pollutes the database with fake claims, skews leaderboards/rankings that depend on claim data, and could be used to make users appear to have claimed amounts they never actually did, potentially affecting downstream systems that rely on this data.

Recommendation

We recommend adding contract address validation to ensure the event comes from the correct contract.

Status: Resolved

6. Missing ValidationPipe and class-transformer enables unvalidated input and injection attacks

Severity: Major

In `zig-claim-api/src/main.ts:16-41`, the NestJS application does not enable the global ValidationPipe, despite defining DTO validation decorators using class-validator. Without the ValidationPipe enabled, the `IsString` and `isNotEmpty` decorators in the DTOs in `claim.dto.ts` are completely ignored, allowing arbitrary malformed input to reach the service layer. This is especially impactful because the API endpoints do not perform any input validation and directly pass the params to database queries through MikroORM.

Recommendation

We recommend enabling the global ValidationPipe as mentioned above.

Status: Resolved

7. Insufficient validation of reward token denomination

Severity: Minor

In the `set_reward_token` function in `zig-claim/src/contract/exec.rs:155-188`, the reward token denomination is only validated for emptiness (line `170`) and `length` (line `174`), but does not validate that it conforms to valid Cosmos SDK denomination format.

Recommendation

We recommend adding validation to ensure the reward token follows valid Cosmos SDK denomination, and has the expected prefix.

Status: Resolved

8. Lack of API rate limiting

Severity: Minor

Throughout the API, there is no rate limiting implemented on any endpoints. This can be taken advantage of by malicious actors to consume unnecessary resources and impact the liveness of the application.

Recommendation

We recommend implementing rate limiting on all the API endpoints.

Status: Resolved

9. Missing validation in admin change allows setting invalid or inaccessible admin addresses

Severity: Minor

In the `change_admin` function in `zig-claim/src/contract/exec.rs:399-422`, the function accepts a new admin address without performing any validation checks. This creates the possibility that the admin can be set to an invalid or inaccessible admin address which could impact core functionality. We classify this as a minor severity because the owner can correct the issue by updating the address to a valid admin address. This issue is also present in the following lines:`zig-claim/src/contract/exec.rs:659` in the `propose_owner` function and in the `add_pauser` function in `zig-claim/src/contract/exec.rs:784`.

Recommendation

We recommend validating the addresses with `deps.api.addr_validate`.

Status: Resolved

10. Case-insensitive address check allows duplicate reward registrations

Severity: Minor

In the `set_user_reward` function in `zig-claim/src/contract/exec.rs:103` the `EVM_WALLET_INFO` checks if `evm_wallet` is already set, but it does not normalize the address first so its possible that the `EVM_WALLET_INFO` map could contain two entries for the same `evm_wallet`. For example *0xAbC...123* and *0xabc...123* represent the same wallet, but this would not return the `EVMWalletAlreadySet` error as expected.

Recommendation

We recommend normalizing all EVM addresses to lowercase before any storage operations or comparisons.

Status: Resolved

11. Token denomination updates invalidate existing claim records

Severity: Minor

In the `set_reward_token` function in `zig-claim/src/contract/exec.rs:155-188`, the admin can change the reward token denomination at any time without checking if users have already been registered or have made claims. This creates a critical accounting vulnerability where the `claimed_amount` stored for each user was denominated in the original token, but future claims will be paid in the new token denomination. We classify this as minor because only the admin can introduce this issue.

Recommendation

We recommend adding a validation check to prevent changing the reward token once any users have been registered.

Status: Resolved

12. Missing validation of `ZigWallet` signer enables reward registration to arbitrary addresses

Severity: Minor

In the `setRewards` function in `zig-claim-api/src/core/claim/claim.service.ts:48-60`, the code verifies that `zigAddress` cryptographically signed the message line 48, but it never validates that `zigAddress` matches `zigWallet` extracted from the message line 31. The function should also make sure that the `zigWallet` is equal to the `zigAddress`.

Recommendation

We recommend ensuring that the `zigWallet` is equal to the `zigAddress`.

Status: Resolved

13. Missing rate limiting creates documentation inconsistency

Severity: Minor

In `zig-claim/src/contract/exec.rs:326-332`, the contract's `claim` function lacks the rate limiting mechanism documented in the README, creating a discrepancy between documented and actual behavior.

The README explicitly states "Rate Limiting: Enforces a minimum interval of 1 hour between claims" (`zig-claim/README.md:83`) while the rate limiting code is entirely commented out.

We classify this issue as minor since this is primarily a documentation inconsistency rather than a security vulnerability. The 180-day vesting period cannot be bypassed.

Recommendation

We recommend either implementing the documented rate limiting by uncommenting the code and defining `MIN_CLAIM_INTERVAL = 3600` (1 hour), or removing the rate limiting documentation from the README if this behavior is no longer desired.

Status: Resolved

14. Missing aggregate allocation tracking in `set_user_reward` could lead to contract insolvency

Severity: Minor

The contract contains a validation gap between per-user allocation limits and total funding limits that enables administrators to create mathematically impossible obligations, guaranteeing insolvency. The issue stems from a 10:1 ratio between funding and allocation limits:

- `MAX_ALLOWED_AMOUNT`: 1 billion tokens
- `MAX_FUND_AMOUNT`: 10 billion tokens

Without aggregate allocation tracking, an administrator can register 11 users each with 1 billion token allocations (11 billion total), while the contract can only be funded with a maximum of 10 billion tokens. Both set_user_reward and fund functions pass their individual validations, creating a 1 billion token shortfall.

As a result, early claimants successfully withdraw their allocations, depleting the contract balance. Late claimants encounter InsufficientContractBalance errors despite having valid on-chain allocations, effectively losing their legitimately allocated rewards.

Recommendation

We recommend implementing aggregate allocation tracking in the contract state to ensure total allocations never exceed total funding. Add a `total_allocated` field to track the sum of all user allocations, and validate in `set_user_reward` that `total_allocated + new_allocation <= total_funded`. Additionally, consider adjusting the constants to maintain a safer ratio or implementing a reservation system to guarantee solvency for all valid allocations.

Status: Acknowledged

15. Missing pagination and result limits on database queries causes unbounded resource consumption

Severity: Minor

Throughout the API, database queries use MikroORM's find method without pagination parameters or result limits, causing queries to load all matching records and return unbounded response payloads. As data grows, this creates performance issues, excessive memory usage, and large response sizes that can timeout browsers or mobile apps.

Specific locations include:

- `zig-claim-api/src/core/rank/rank.service.ts:12-15`
- `zig-claim-api/src/core/rank/rank.service.ts:86-163`
- `zig-claim-api/src/core/rank/rank.service.ts:288-290`

For example, the leaderboard could return tens of thousands of entries, forcing clients to receive megabytes of JSON data even when they only need top 10 results.

Recommendation

We recommend implementing pagination with limit and offset parameters on all queries. Add reasonable defaults (e.g., limit=100) and maximum limits (e.g., max limit=1000) to prevent resource exhaustion. For leaderboard endpoints, consider returning only a subset by default (e.g., top 100) and allow clients to request specific pages if needed.

Status: Acknowledged

16. Lack of input validation in multiple instances

Severity: Minor

Throughout the API, query parameters from user input (wallet addresses, transaction hashes, etc.) are passed directly to MikroORM's find method without validation. While MikroORM uses parameterized queries which prevent SQL injection, the functions do not validate expected formats or length constraints before executing database queries. It is best practice to implement defense in depth practices to prevent malformed or malicious user input from impacting the database.

An attacker could potentially send extremely long strings or malformed data that can cause database query performance degradation and waste server resources. Specific locations include:

- `zig-claim-api/src/core/claim/claim.service.ts:26-28`
- `zig-claim-api/src/core/claim/claim.service.ts:65-70`
- `zig-claim-api/src/core/claim/claim.service.ts:74`
- `zig-claim-api/src/core/rank/rank.service.ts:17-19`

Recommendation

We recommend implementing additional input validations for the abovementioned parameters which have well-known formats and length.

Status: Resolved

17. `parseMessage` regex should implement max message length validation

Severity: Informational

The `parseMessage` function in `zig-claim-api/src/core/claim/claim.helper.ts:1` does use exact length qualifiers to ensure addresses are safely parsed but even with safe regexes, extremely large inputs can slow down operations, consume excessive memory and impact application performance.

Recommendation

We recommend implementing a message length validation as an effective defense in depth measure.

Status: Resolved

18. Remove unused code blocks

Severity: Informational

The codebase has multiple instances of commented-out code blocks. Maintaining dead code that is not actively used impacts the maintainability and readability of the codebase. These include:

- `zig-claim/src/contract/exec.rs:111-133`
- `zig-claim/src/contract/exec.rs:151-152`
- `zig-claim-api/src/core/claim/claim.service.ts:54-55`
- `zig-claim/src/contract/exec.rs:327-332`.

Recommendation

We recommend removing all commented-out code blocks from production code.

Status: Resolved

19. Missing check to prevent pausing an already paused contract

Severity: Informational

In the `pause` function in `zig-claim/src/contract/exec.rs:200-230`, there is no validation to prevent pausing a contract that is already paused. While this does not create a security vulnerability, it allows redundant pause operations that unnecessarily update the `pause_reason` and `pause_timestamp` fields. This could be confusing in operational scenarios where multiple pausers might attempt to pause an already-paused contract, or where monitoring systems track pause events.

Recommendation

We recommend returning an error if the contract is already paused.

Status: Resolved

20. Hardcoded gas price prevents flexible fee adjustment and creates operational risk

Severity: Informational

In the `sendSetUserRewardsTx` function in `zig-claim-api/src/lib/zigchain.ts:25`, the gas price is hardcoded to `0.025uzig` instead of being configurable via environment variables or dynamic pricing. While this specific price may be appropriate for current network conditions, hardcoding gas prices creates the risk that user transactions may not succeed.

Recommendation

We recommend setting an adjustable gas price with an environment variable and then setting the default fallback value to `0.025uzig`.

Status: Resolved

21. Unused `getClaimsByWalletAddress` function

Severity: Informational

In the `ClaimService` in `zig-claim-api/src/core/claim/claim.service.ts:25-28`, the `getClaimsByWalletAddress` function is defined but never called in the codebase. This impacts the maintainability of the service.

Recommendation

We recommend implementing this code or removing it from the codebase.

Status: Resolved

22. Unregistered `IpLoggerMiddleware`

Severity: Informational

In `zig-claim-api/src/core/claim/claim.service.ts:25-28`, the `IpLoggerMiddleware` is defined but never registered in any NestJS module. As a result, IP addresses are not being logged for incoming requests, which may impact security monitoring and audit capabilities.

Recommendation

We recommend either registering this middleware in the appropriate module configuration to enable IP logging, or removing the unused code if IP logging is not required.

Status: Resolved

23. Typo in CORS configuration

Severity: Informational

In the domain configuration in `zig-claim-api/src/lib/security/config.ts:13`, the `LOCAL` environment array contains `"http://localhost::4200"` with a double colon between the hostname and port number. This malformed URL is then passed to the CORS configuration in `main.ts:32` where all domains are whitelisted. This will cause requests to fail.

Recommendation

We recommend fixing the typo in line `13` by changing `http://localhost::4200` to `http://localhost:4200` with a single colon.

Status: Resolved