Security Audit Report

Valdora

v1.2

September 28, 2025

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

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 LIQUIDZIG LTD to perform a security audit of Audit of the Valdora protocol smart contracts.

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/Liquid-Zig/new-stzig-contracts
Commit	`5fd9d3bf7a97daee3630e5487f6a6384013f3e21`
Scope	All contracts were in scope.
Fixes verified at commit	`9abdd76a5735a3894446d80366835077819c18ed` 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

StZIG is a liquid staking protocol built on ZigChain using CosmWasm that enables users to stake their native ZIG tokens while maintaining liquidity through stZIG receipt tokens. The protocol manages a multi-ledger architecture where the main staker contract coordinates deposits across multiple ledger contracts that interface with different validators, implementing sophisticated logic for rewards distribution, slashing detection, and emergency withdrawals. Users can deposit ZIG to receive stZIG tokens at a dynamically calculated exchange rate that appreciates over time as staking rewards accumulate, while the protocol handles unbonding periods, manages deposit and reward buffers, enforces role-based access control for administrative functions, and provides emergency mode capabilities to protect user funds during critical events.

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	The protocol implements complex state management across multiple interacting contracts.
Code readability and clarity	Medium-High	-
Level of documentation	High	-
Test coverage	Medium-High	Test coverage is at 78.67%.

Summary of Findings

No	Description	Severity	Status
1	Unbonding request ID collision leads to loss of funds	Critical	Resolved
2	Reward overwriting leads to loss of accumulated rewards	Critical	Resolved
3	Incorrect total supply calculation affects `stzig` redemption rate	Critical	Resolved
4	Withdrawal operations enable DoS attacks on core protocol functions	Major	Resolved
5	Incorrect fee minting calculation treats `uzig` as `stzig` overpaying the treasury	Major	Resolved
6	Ledgers with unbonding requests excluded from reconciliation	Major	Resolved
7	Duplicate withdrawal requests removed leading to user fund loss	Major	Resolved
8	Incorrect available bonded amount calculation causes underflow	Major	Resolved
9	Slash calculation excludes unbonding amounts leading to inaccurate loss tracking	Major	Partially Resolved
10	Unbonding requests not updated after slash detection lead to discrepancies in fund calculations	Major	Resolved
11	Buffer funds lost when no available ledgers for bonding	Major	Resolved
12	Slash Factor implementation contains mathematical error preventing proper loss distribution	Major	Resolved
13	Reward parsing defaults to zero for non-uzig denominations	Minor	Acknowledged
14	Unsafe arithmetic operations throughout codebase	Minor	Resolved
15	Token creation fails when ZIGChain's `DenomCreationFee` is not zero	Minor	Resolved
16	Gas consuming `get_ledgers` function may cause	Minor	Resolved

	DoS		
17	Reward double counting due to unreset current rewards	Minor	Resolved
18	Execution sequence safety concerns	Minor	Acknowledged
19	Insufficient balance causes `Sync` failure	Minor	Resolved
20	Emergency withdrawal may fail due to insufficient balance	Minor	Resolved
21	Missing input validation checks could accept invalid parameters in multiple functions	Minor	Resolved
22	Centralization risks	Minor	Acknowledged
23	Missing token burn amount validation	Minor	Resolved
24	Missing maximum ledger limit could eventually cause issues with operations that iterate over all ledgers	Informational	Acknowledged
25	Redundant ledger settings retrieval	Informational	Resolved
26	Unclear distinction between admin and user functions during pause	Informational	Resolved
27	Potentially non-unique reply IDs using enum variants	Informational	Resolved
28	Use of magic numbers decreases maintainability	Informational	Resolved
29	Inconsistent protocol modifier naming	Informational	Resolved
30	Misleading `current_supply` variable naming and redundant minting cap check	Informational	Resolved
31	Incorrect comment about contract owner check	Informational	Resolved
32	Admin overwriting in ledger registration may cause confusion	Informational	Resolved
33	Misleading withdrawal request ID and inaccurate completion time	Informational	Resolved
34	Inconsistency in paused contract checks may lead to confusion	Informational	Resolved
35	Unused variables and functions	Informational	Resolved
36	Incomplete documentation of default protocol modifiers	Informational	Resolved

37	Inefficient filtering of available ledgers for unbonding	Informational	Resolved
38	Redundant expensive `get_ledgers` call in `Sync` function	Informational	Resolved
39	Incorrect documentation for withdraw completed unbonding function	Informational	Resolved
40	Migration functions cannot pass custom parameters	Informational	Resolved
41	Redundant paused status checks	Informational	Resolved
42	Contracts should implement a two-step ownership transfer	Informational	Resolved
43	Burn function address parameter is misleading	Informational	Resolved
44	Confusing event logs	Informational	Resolved
45	Debug code in production	Informational	Resolved
46	Missing documentation for multiple functionalities	Informational	Resolved
47	Lack of event emissions prevents off-chain monitoring and integration	Informational	Resolved

Detailed Findings

1. Unbonding request ID collision leads to loss of funds

Severity: Critical

In `contracts/ledger/src/contract/exec.rs:137-146`, the value of `current_requests` is taken from `UNBONDING_REQUEST_COUNT`, which represents the current number of unbonding requests up to `max_unbonding_requests`. However, when an entry is removed from `UNBONDING_REQUESTS` during reconciliation in lines `408-409`, the IDs of existing requests are maintained while only the count is decremented.

This creates a scenario where new unbonding requests can overwrite existing ones. For example, if there are 3 unbonding requests with IDs 1, 2, and 3, and request ID 1 is removed during `execute_reconcile`, the `UNBONDING_REQUEST_COUNT` becomes 2. The next `execute_unbond` call will use `current_requests + 1` (which equals 3) as the new request ID, overwriting the existing unbonding request with ID 3.

This results in the loss of the overwritten unbonding request and the associated funds, as users will be unable to claim their unbonded tokens when the original request reaches completion time.

Recommendation

We recommend implementing an incrementing ID system with a `LAST_ID` counter that tracks the highest ID used and never reuses IDs. When implementing this fix, modify the loop in lines `397-405` to iterate only over existing requests rather than a sequential range from 1 to `LAST_ID`, as the ID space could grow indefinitely and cause performance issues.

Status: Resolved

2. Reward overwriting leads to loss of accumulated rewards

Severity: Critical

In `contracts/ledger/src/contract.rs:143-148`, after successful `Bond`, `Unbond`, `Redelegate`, and `Reconcile` calls, some rewards may be withdrawn on the ledger contract. However, at each of these calls, the `CURRENT_REWARD` is overwritten instead of being incremented, meaning that the value of `CURRENT_REWARD` is not the actual amount of rewards since last time it was synchronized in staker contract.

The staker contract retrieves rewards through the `execute_distribute_rewards_and_fees` function in `contracts/staker/src/contract/exec.rs:1079-1082`, which reads the `current_reward` value from each ledger. When multiple staking operations occur

before this distribution call, only the rewards from the last operation are preserved, while all previous rewards are lost.

This results in significant loss of user rewards. For instance, if a `Redelegate` operation is issued before calling `Reconcile`, the `Reconcile` call would withdraw rewards and overwrite `CURRENT_REWARD`. A subsequent call to `DistributeRewardsAndFees` would only account for the rewards from the `Reconcile` operation, ignoring all rewards from the `Redelegate`.

Recommendation

We recommend increasing `CURRENT_REWARD` at each successful reward withdrawal instead of overwriting it. The staker contract should reset this counter to zero after retrieving the accumulated rewards during distribution.

Status: Resolved

3. Incorrect total supply calculation affects `stzig` redemption rate

Severity: Critical

In `contracts/staker/src/contract/exec.rs:421-426` and `640-645`, and `contracts/staker/src/contract/query.rs:111-116` and `181-186`, the functions that calculate the conversion between `stzig` and `uzig` use the `TotalSupply` query and assume this represents the current total supply of the `stzig` token.

However, this query returns the total amount of tokens minted, which is not reduced after burning tokens. This is not the actual circulating supply and cannot be used to calculate the correct `stzig` redemption rate. For instance, the `stzig` -> `uzig` redemption rate will incorrectly increase after users redeem (burn) their `stzig` tokens.

This results in a fundamentally broken model where the exchange rate does not reflect the actual token supply, potentially leading to significant financial losses for the protocol. Users may receive incorrect amounts during redemption, and the protocol's value calculations will be persistently inaccurate.

Additionally, the `query_total_supply` functions in `contracts/staker/src/contract/query.rs:175-192` and `contracts/token/src/contract/query.rs:25-34` are incorrectly labeled and may bring confusion to developers who would expect querying the total supply. Instead, they receive the minted information.

Recommendation

We recommend using the actual circulating supply (total minted minus total burned) instead of the cumulative minted amount when calculating the redemption rate. Implement proper tracking of burned tokens or query mechanisms that return the true circulating supply.

Status: Resolved

4. Withdrawal operations enable DoS attacks on core protocol functions

Severity: Major

Multiple functions in the staker contract load all withdrawal requests into memory, creating scalability bottlenecks and DoS vulnerabilities.

In `contracts/staker/src/contract/exec.rs:1187-1195`, the `get_resolvable_withdrawals` function retrieves all withdrawal requests into memory, then sorts them all by timestamp using `sort_by` which has `O(n log(n))` complexity. This function is called in the `Sync` message, and as the number of withdrawal requests grows, it may eventually exceed gas limits and become unusable.

Similarly, in `contracts/staker/src/contract/exec.rs:2204-2206`, the `execute_distribute_losses` function loads all withdrawal requests into memory. This is vulnerable to denial-of-service attacks where malicious users can create large numbers of withdrawal requests. Even though there is a limit per user, an attacker can circumvent this by distributing `stzig` tokens to multiple addresses and creating withdrawal requests from each address. For example, if the limit is 10 withdrawals per user and the attacker has 10 `stzig` tokens (10,000,000 units), they could create 1 million addresses with 10 `ustzig` each and initiate 1 million withdrawal requests, severely bloating storage and making the function prohibitively expensive to execute.

This creates scalability bottlenecks and potential DoS vectors that could render both the withdrawal resolution process and loss distribution process inoperable, effectively breaking core protocol functionality.

Recommendation

We recommend redesigning the storage structure to use two separate stores: `WITHDRAWAL_QUEUE: Map<u64, WithdrawalRequest>` sorted by incremental ID for efficient processing, and `USER_WITHDRAWAL_REQUEST_IDS: Map<String, Vec<u64>>` for per-user access. This approach eliminates the need to load all requests into memory and sort them, providing `O(1)` access to the earliest withdrawal requests.

Additionally, we recommend setting up a minimum withdrawal amount requirement to make DoS attacks economically unfeasible, such as requiring at least 1 `stzig` (1,000,000 `ustzig`) for one withdrawal request.

Status: Resolved

5. Incorrect fee minting calculation treats `uzig` as `stzig` overpaying the treasury

Severity: Major

In `contracts/staker/src/contract/exec.rs:1109-1121`, after calculating the `fees_to_mint` using the `fees_percentage` setting, those fees are minted as `stzig` tokens. However, while `fees_to_mint` is calculated as a number of `uzig` tokens, this value is used directly as the number of `stzig` tokens to mint without proper conversion.

This creates a significant discrepancy in the fee collection mechanism, as `uzig` and `stzig` have different values based on the exchange rate. The protocol will mint incorrect amounts of `stzig` for treasury fees, overpaying the treasury.

Recommendation

We recommend using the `calculate_stzig_price` function to convert the uzig-denominated `fees_to_mint` value to the appropriate amount of `stzig` tokens to mint for the treasury, ensuring accurate fee collection based on the current exchange rate.

Status: Resolved

6. Ledgers with unbonding requests excluded from reconciliation

Severity: Major

In `contracts/staker/src/contract/exec.rs:780-784`, the `execute_reconcile_ledgers` function filters ledgers to include only those that have bonded amounts when determining which ledgers need reconciliation. The reconciliation process handles clearing completed unbonding requests, withdrawing rewards, and checking for slashing events.

However, this filtering excludes ledgers that have no bonded tokens but still have active unbonding requests. These ledgers will not be processed during reconciliation, leaving them unreconciled and potentially failing to detect slashing events that could affect their unbonding tokens.

This creates an inconsistent state where some ledgers with unbonding requests are not properly maintained, and slashing events affecting unbonding tokens may go undetected, leading to inaccurate accounting and potential loss of funds.

Recommendation

We recommend including ledgers with unbonding amounts in the reconciliation process by modifying the filter condition to also include ledgers that have non-zero unbonding amounts, ensuring comprehensive reconciliation of all active ledgers.

Status: Resolved

7. Duplicate withdrawal requests removed leading to user fund loss

Severity: Major

In `contracts/staker/src/contract/exec.rs:1262-1264`, when processing resolvable withdrawal requests, requests are removed by matching `timestamp` and `share_value`. However, these values are not guaranteed to be unique, as a user could create a transaction with multiple `Redeem` messages with the same amount, resulting in multiple withdrawal requests with identical `timestamp` and `share_value`.

When only one of these duplicate requests can be resolved, the removal logic using retain will eliminate all matching requests, including unprocessed ones. This results in users losing their funds for withdrawal requests that were removed but never actually processed or paid out.

This creates a vulnerability where users can permanently lose their staked funds if they submit multiple withdrawal requests with the same parameters, as the system cannot distinguish between processed and unprocessed requests with identical matching criteria.

Recommendation

We recommend using unique IDs in `USER_WITHDRAWAL_REQUESTS` to ensure that only the specific processed request is removed, preventing accidental removal of duplicate but unprocessed withdrawal requests.

Status: Resolved

8. Incorrect available bonded amount calculation causes underflow

Severity: Major

In `contracts/staker/src/contract/exec.rs:1349`, `1363`, `1369-1370`, and `1383`, the `unbonding_amount` is subtracted from `bonded_amount` to calculate the available bonded amount. However, `unbonding_amount` represents tokens that are already in the

unbonding process and is not included in `bonded_amount`, making this subtraction logically incorrect.

Additionally, if `unbonding_amount` is greater than `bonded_amount`, this calculation will result in an integer underflow, causing the transaction to produce incorrect results.

This affects the ledger selection logic for unbonding operations within the `Sync` message execution.

Recommendation

We recommend using `bonded_amount` directly as the available bonded amount since it already represents the tokens that are currently bonded and available for unbonding operations, without subtracting the separate `unbonding_amount`.

Status: Resolved

9. Slash calculation excludes unbonding amounts leading to inaccurate loss tracking

Severity: Major

In `contracts/ledger/src/contract/exec.rs:433-438`, when reconciling slash events, the loss is calculated as `stored_bonded_amount - queried_bonded_amount`, representing the difference between the expected bonded amount and the actual bonded amount after slashing.

However, slash events affect both bonded and unbonding tokens, not just bonded tokens. By only considering the bonded amount in the loss calculation, the system fails to account for slashed unbonding tokens, resulting in an underestimated slash amount stored in `SLASH_TODAY`.

Furthermore, this inaccurate slash amount is used in the staker contract's `execute_distribute_losses` function in `contracts/staker/src/contract/exec.rs:2189` to calculate the `loss_percentage` by comparing it with `FUNDS_RAISED_BALANCE`. Since `FUNDS_RAISED_BALANCE` is not synchronized with the bonded amount and is updated independently through `Deposit` or `Redeem` operations, this creates additional inconsistencies in loss percentage calculations.

The inaccurate loss tracking leads to incorrect distribution of losses to users and potentially unfair allocation of remaining funds.

Recommendation

We recommend that the staker contract maintain accurate tracking of expected amounts across all states (bonded, unbonding, and unbonded awaiting withdrawal) for each ledger and compare these with the actual queried amounts. Since slashing can occur at any time and affect tokens in different states, the loss calculation should account for the total difference

across all these states rather than relying on `FUNDS_RAISED_BALANCE` for percentage calculations.

Status: Partially Resolved

Note: Client implemented the three-bucket tracking system as recommended. Due to CosmWasm/ZIGChain limitations, Client advised that the unbonding amounts cannot be queried directly, so slash detection occurs at maturity. This is acceptable given the technical constraints.

Important: The contract's unbonding period must match the chain's actual unbonding period. Any mismatch will cause incorrect slash detection timing. Client should implement monitoring and update mechanisms for potential unbonding period changes. Please note that in case of an unbonding period change, the existing unbonding requests should keep the completion time as calculated initially, and the new unbonding period should only apply to new unbonding requests.

10. Unbonding requests not updated after slash detection lead to discrepancies in fund calculations

Severity: Major

In `contracts/ledger/src/contract/exec.rs:444-446`, when a loss is detected during `execute_reconcile`, the `BONDED_AMOUNT` is updated to reflect the actual bonded amount after slashing. However, the `UNBONDING_REQUESTS` are left untouched despite being potentially affected by the same slash event.

This creates inconsistency in the contract state where the bonded amount reflects post-slash reality while unbonding requests maintain their pre-slash values. The `UnbondingAmount` query, which is used by the staker contract's `get_ledger` function and subsequently in multiple functions, returns incorrect values based on outdated unbonding request amounts.

This leads to inaccurate accounting in the staker contract and potential discrepancies in fund calculations, as the system believes there are more unbonding tokens than actually exist after slashing.

Recommendation

We recommend updating the `UNBONDING_REQUESTS` with accurate values from existing undelegations.

Status: Resolved

11. Buffer funds lost when no available ledgers for bonding

Severity: Major

In `contracts/staker/src/contract/exec.rs:1514-1518`, the `available_ledgers_for_bonding` is retrieved by filtering ledgers where the `unbonding_requests_count` is less than `max_unbonding_requests`. It is possible that no ledgers meet this requirement, rendering `available_ledgers_for_bonding` an empty vector.

An empty `available_ledgers_for_bonding` would skip the bonding logic in lines `1522-1540`, resulting in no `LedgerExecMsg::Bond` messages being executed. However, even when no bonding occurs, both `BUFFER_DEPOSITS` and `BUFFER_REWARDS` are reset to zero in lines `1571-1572`, causing the loss of buffered funds that should have been bonded.

This creates a scenario where user deposits and accumulated rewards are lost (not accounted for) if all ledgers have reached their maximum unbonding request limits, leaving these funds in the staker contract's balance.

Recommendation

We recommend not requiring available unbonding slots for bonding operations, as bonding should always be available regardless of unbonding capacity.

Additionally, we recommend not resetting the `BUFFER_DEPOSITS` and `BUFFER_REWARDS` stores if no ledger is available for bonding, and instead increasing `BUFFER_DEPOSITS` with `unresolved_amount` so it can be used in the next `Sync` attempt.

Status: Resolved

12. Slash Factor implementation contains mathematical error preventing proper loss distribution

Severity: Major

In `contracts/staker/src/contract/exec.rs`:

- Line 3002 in `execute_distribute_losses` calculates `loss_ray = (loss_percentage / FEE_PERCENTAGE_DENOMINATOR) * SLASH_FACTOR_RAY`. When `loss_percentage < 10000` (100%), integer division causes the result to be 0, completely preventing slash distribution to withdrawal requests.
- Lines 1683-1684 in `process_withdrawal` have inefficient store loads of `EPOCH` and `SLASH_FACTORS` inside the loop, loading the same values repeatedly.

Recommendation

We recommend:

- Fixing the calculation order multiplying by `SLASH_FACTOR_RAY` before dividing by `FEE_PERCENTAGE_DENOMINATOR`: `loss_ray = (loss_percentage * SLASH_FACTOR_RAY) / FEE_PERCENTAGE_DENOMINATOR`.
- Moving `EPOCH.load` and `SLASH_FACTORS.load` outside the loop in `process_withdrawal` since these values do not change during iteration
- Implementing proper loss calculation across all states (bonded, unbonding, unbonded) as mentioned in issue #9.

Status: Resolved

13. Reward parsing defaults to zero for non-uzig denominations

Severity: Minor

In `contracts/ledger/src/contract.rs:108-129`, the code retrieves the first attribute with key `amount` from the `withdraw_delegator_reward` event. The implementation assumes this amount is always denominated in `uzig` tokens by using `trim_end_matches("uzig")` to extract the numeric value.

However, staking rewards can be distributed in multiple denominations if tokens are sent to the fee collector. If the first amount attribute in the event is not `uzig`, the `trim_end_matches("uzig")` operation will fail to extract a valid numeric value, causing the parsing to fail and the reward amount to default to zero in line `124`.

We classify this issue as minor, as ZIGChain currently disables sending tokens to the fee collector, although this configuration may change in the future to allow other token denoms. If this occurs, legitimate staking rewards in `uzig` denomination may be ignored and not credited.

Recommendation

We recommend modifying the reward parsing logic to handle multiple staking reward denominations properly by specifically extracting the `uzig` denomination. This will ensure the system remains robust if ZIGChain enables multi-denomination rewards in the future.

Status: Acknowledged

14. Unsafe arithmetic operations throughout codebase

Severity: Minor

A number of unsafe additions, subtractions, multiplications and divisions are present throughout the codebase. While these operations are probably safe most of the time given

the context and typical values, they pose a risk of integer overflow or underflow that could lead to unexpected behavior or contract failures.

Using direct arithmetic operators (`+`, `-`, `*`, `/`) instead of overflow-safe functions means that potential arithmetic errors would go undetected and could result in incorrect calculations or contract panics in edge cases.

Recommendation

We recommend using overflow-safe functions such as `checked_add`, `checked_sub`, `checked_mul`, and `checked_div` instead of direct arithmetic operators when possible. These functions return errors when overflow or underflow is detected, allowing for proper error handling and preventing silent calculation errors.

Status: Resolved

15. Token creation fails when ZIGChain's `DenomCreationFee` is not zero

Severity: Minor

The token contract uses ZIGChain's token `factory` module to create the `stzig` token. The `factory` module comes with a `DenomCreationFee` parameter, which is consumed whenever someone calls `CreateDenom`.

However, in `contracts/staker/src/contract/exec.rs:126-135`, the staker contract instantiates the token contract but does not send any funds with the `instantiate` message. The funds field in line `131` is set to an empty vector.

This means that if `DenomCreationFee` is not set to zero, the `RegisterToken` message will fail due to insufficient funds to pay the required denomination creation fee, preventing the token contract from initializing properly.

Recommendation

We recommend forwarding funds sent to the `RegisterToken` function in the staker contract to the `WasmMsg::Instantiate` message in line `131` to cover the potential `DenomCreationFee` required by ZIGChain's `factory` module.

Otherwise, if ZIGChain's `DenomCreationFee` is intended to be zero on mainnet, we recommend documenting that this is a prerequisite for the deployment of the token contract.

Status: Resolved

16. Gas consuming `get_ledgers` function may cause DoS

Severity: Minor

In `contracts/staker/src/contract/exec.rs:909-931`, the `get_ledgers` function is very gas consuming as it browses all registered ledgers, and for each of them, the call to `get_ledger` runs 7 different contract queries in sequence.

We classify this issue as minor because this function is only called by admin-accessible functions, and the admin has control over the number of ledgers.

However, as the number of ledgers increases, this function could become prohibitively expensive to execute, potentially causing transaction failures due to gas limits.

Recommendation

We recommend limiting the smart contract queries to what is actually required by adding an argument to the `get_ledgers` function that would instruct which queries are needed for each specific call. This would allow avoiding querying all 7 queries each time when only a subset of the information is needed.

Additionally, implementing a maximum ledger limit would help prevent this function from becoming unusably expensive.

Status: Resolved

17. Reward double counting due to unreset current rewards

Severity: Minor

In `contracts/staker/src/contract/exec.rs:1079-1132`, the `execute_distribute_rewards_and_fees` function retrieves the `total_rewards` value by summing the `current_reward` from all ledgers and increases `FUNDS_RAISED_BALANCE` accordingly. However, the function does not reset the `current_reward` values on the individual ledgers after processing.

This creates a vulnerability where calling `execute_distribute_rewards_and_fees` multiple times without actually claiming the rewards from the ledger contracts will result in double counting of the same rewards. Each subsequent call will re-add the same reward amounts to `FUNDS_RAISED_BALANCE`, inflating the total funds and affecting all subsequent calculations that depend on this value, such as the `stzig` redemption rate.

This leads to incorrect protocol accounting, inflated exchange rates, where rewards are counted multiple times without being properly consumed.

We classify this issue as minor, since it can only be called by the admin.

Recommendation

We recommend resetting the `current_reward` values of all ledgers to zero after successfully processing the rewards in `execute_distribute_rewards_and_fees` to prevent double counting in subsequent function calls.

Status: Resolved

18. Execution sequence safety concerns

Severity: Minor

In `contracts/staker/README.md:23`, the documentation describes a mandatory execution sequence that must be run every 24 hours. The system state is not safe after individual functions are executed and requires the complete sequence to be executed in order to maintain a valid state.

This design creates vulnerability to state inconsistencies, particularly with stores like `SLASH_TODAY` that only keep the last value. If an issue occurs in one step (such as `DistributeLosses`) and `ReconcileLedger` is called again, it will overwrite the value in `SLASH_TODAY`, potentially causing data loss and state corruption.

The current automation scripts shown in lines `150-265` do not implement proper error handling that would stop execution upon encountering failures, which could lead to partially completed sequences and invalid contract states if not monitored appropriately.

Recommendation

We recommend ensuring that all contract states remain valid regardless of execution order, eliminating the strict dependency on sequential execution.

Additionally, implement proper error handling in automation scripts where any error in one function results in stopping the entire process, requiring manual intervention and fix before allowing restarts.

Status: Acknowledged

19. Insufficient balance causes `Sync` failure

Severity: Minor

In `contracts/staker/src/contract/exec.rs:1453` and `1457`, if `buffer_deposits + buffer_rewards > balance`, then `available_funds` becomes zero, resulting in `unresolved_amount` also being zero. This leads to `remaining_amount = buffer_deposits + buffer_rewards`, which exceeds the available balance.

When the bonding logic in lines `1522-1540` attempts to bond `remaining_amount` to ledgers, it will fail due to insufficient funds since `remaining_amount > balance`. This causes the entire `Sync` call to fail, preventing the protocol from processing any synchronization operations until the balance issue is resolved.

This scenario can occur when there are accumulated buffer deposits and rewards but insufficient actual token balance in the contract to fulfill the bonding operations.

Recommendation

We recommend adding balance validation before attempting to bond funds, and adjusting the bonding amount to not exceed the available contract balance to prevent `Sync` failures due to insufficient funds.

Status: Resolved

20. Emergency withdrawal may fail due to insufficient balance

Severity: Minor

In `contracts/staker/src/contract/exec.rs:2520-2526`, the emergency withdrawal function attempts to transfer the entire `funds_raised` amount to the target address. However, there is no guarantee that the staker contract balance contains sufficient `uzig` tokens to cover this amount, even after waiting to unstake all delegations from all ledgers.

Discrepancies can arise due to rounding errors, slashing events, and other factors that may cause the actual contract balance to be less than the recorded `funds_raised` amount. This would cause the emergency withdrawal to fail when attempting to transfer more tokens than are available.

Recommendation

We recommend withdrawing the actual available balance instead of the `funds_raised` amount, and updating `FUNDS_RAISED_BALANCE` accordingly to reflect the actual amount transferred during emergency withdrawal.

Status: Resolved

21. Missing input validation checks could accept invalid parameters in multiple functions

Severity: Minor

Multiple functions across the codebase lack proper validations:

- In `contracts/staker/src/contract/exec.rs:660`, in `execute_update_minting_cap` – no upper bound check.
- In `contracts/staker/src/contract/exec.rs:991`, in `execute_update_fees_percentage` – no check if fee > 10000 (100%).
- In `contracts/ledger/src/contract/exec.rs:278`, in `execute_update_ledger_settings` – minimal validation, no upper bounds checks.

Recommendation

We recommend implementing comprehensive validation:

- Percentage values: ensure <= 10000 (100%),
- Caps and limits: reasonable upper bounds,
- All numeric inputs: range validation

Also, make sure to call one function for validation from where it is applicable, instead of duplicating checks in different parts of the codebase.

Status: Resolved

22. Centralization risks

Severity: Minor

The protocol has critical centralization risks that give the admin complete control over user funds:

- Single admin model: All contracts use single admin without multi-sig.
- No timelock: Critical operations execute immediately without delay.
- No governance: All decisions are centralized to admin.
- Trust in ledger contracts: Staker trusts all ledger-reported values without verification.

Recommendation

We recommend implementing multi-signature controls, timelocks for critical operations, and eventually transitioning to decentralized governance. These centralization aspects should be clearly documented for users.

Status: Acknowledged

23. Missing token burn amount validation

Severity: Minor

In `contracts/token/src/contract/exec.rs:160`, the `execute_burn` function does not validate that the amount of tokens sent matches the requested burn amount.

The risk is burning less than the tokens sent, as ZIGChain's BurnTokens can only burn what's in the balance of the token contract. If the owner sends more tokens than the burn amount parameter, the excess tokens remain in the contract.

Recommendation

We recommend validating that the sent amount matches the burn amount parameter.

Status: Resolved

24. Missing maximum ledger limit could eventually cause issues with operations that iterate over all ledgers

Severity: Minor

In `contracts/staker/src/contract/exec.rs:175-217` in `execute_register_ledger` function, while ledger creation is now manual (admin-only) which addresses the original gas exhaustion issue, no hard limit on the total number of ledgers exists.

Unbounded growth could eventually cause issues with operations that iterate over all ledgers.

Recommendation

We recommend implementing a maximum of 3 ledgers per validator (that they want to include in the LST set), so they can have one unbond per validator per day if they want (7 unbondings per ledger x 3 ledgers per validator = 21 unbondings per validator).

Status: Acknowledged

25. Redundant ledger settings retrieval

Severity: Informational

In `contracts/ledger/src/contract/exec.rs:113`, `ledger_settings` is retrieved from the `LEDGER_SETTINGS` store. It is then retrieved again in line `120`, while it has not been modified since the first retrieval.

This creates unnecessary processing overhead and storage reads that could be avoided by reusing the already loaded value.

Recommendation

We recommend removing line `120` to avoid unnecessary processing and use the `ledger_settings` variable that was already loaded in line `113`.

Status: Resolved

26. Unclear distinction between admin and user functions during pause

Severity: Informational

In `contracts/ledger/src/contract.rs:30-73`, some functions are allowed even when the contract is paused while others are blocked, with comments distinguishing between "admin functions" and "user functions". However, all functions require admin privileges to execute, making the distinction between "user functions" that are blocked when paused and "admin functions" that are always allowed unclear.

The functions `Bond`, `Unbond`, `Reconcile`, and `Redelegate` are blocked when paused and labeled as "user functions", while `UpdateAdmin`, `UpdateLedgerSettings`, `WithdrawBalance`, and `ResetSlashToday` are always allowed and labeled as "admin functions". Since all functions are admin-gated, the rationale for this categorization is not apparent from the code.

Recommendation

We recommend clarifying the reasoning behind why certain admin functions are blocked during pause while others are not. Consider adding documentation or comments explaining the business logic for this distinction to improve code maintainability and reduce confusion for future developers.

Status: Resolved

27. Potentially non-unique reply IDs using enum variants

Severity: Informational

In `contracts/ledger/src/contract.rs:96-97`, the code uses `crate::msg::ReconcileReply::WithdrawRewardsSuccess {} as u64` which evaluates to `0`, as it is the first variant of the `ReconcileReply` enum. This approach creates a potential issue where if additional enums are introduced for reply IDs, their first variants would also evaluate to `0`, causing ID collisions.

When multiple enums are used for different types of reply handling, this could break the reply matching logic as the same numerical ID could correspond to different reply types, leading to incorrect message processing.

Recommendation

We recommend using explicit numerical constants for reply IDs rather than relying on enum variant positions, or clarifying that only one reply enum is to be used.

Status: Resolved

28. Use of magic numbers decreases maintainability

Severity: Informational

In `contracts/staker/src/contract/init.rs:87`, `contracts/staker/src/contract/exec.rs:1000` and `1104`, the number literal `10000` is used to represent the fee percentage denominator.

Using such "magic numbers" goes against best practices as they reduce code readability and maintenance as developers are unable to easily understand their use and may make inconsistent changes across the codebase.

Recommendation

We recommend creating named constants for all instances of magic values outlined above, such as `const FEE_PERCENT_DENOMINATOR: u64 = 10000;` to improve code readability and maintainability.

Status: Resolved

29. Inconsistent protocol modifier naming

Severity: Informational

In `contracts/staker/src/contract/init.rs:138-143`, some protocol modifiers are not named consistently with other protocol modifiers. While the actual functions start with the prefix `execute_`, the protocol modifier names for `add_protocol_modifiers`, `remove_protocol_modifiers`, `add_role`, `remove_role`, `add_user`, and `remove_user` do not include this prefix.

This inconsistency can create maintenance issues when updating authorization logic.

Recommendation

We recommend updating the protocol modifiers in `contracts/staker/src/contract/init.rs:138-143` to use consistent naming with

the `execute_` prefix, and updating the corresponding calls to `assert_authorization` in each `execute` function to ensure that permissions are still checked correctly.

Status: Resolved

30. Misleading `current_supply` variable naming and redundant minting cap check

Severity: Informational

In `contracts/token/src/contract/exec.rs:67`, `current_supply` is initialized with the minted information of the queried `denom_info`. This is confusing, since "current supply" typically refers to the amount of tokens in circulation, while minted informs about the total number of tokens ever minted. Basically, the typical definition of `current_supply` would be equal to `minted - burned`.

Furthermore, the check at lines `71-77` that `new_supply` should not be greater than `minting_cap`, while correct, is redundant with the underlying ZIGChain `factory` module, which already runs this check before allowing the minting of new tokens.

This creates confusion about the actual meaning of the variable and adds unnecessary validation that duplicates chain-level checks.

Recommendation

We recommend renaming `current_supply` to reflect that its value is actually the number of tokens minted so far. Also, it is possible to remove the check lines `71-77`, unless it is preferred to return the `ExceedsMintingCap` error in case of exceeding the cap.

Status: Resolved

31. Incorrect comment about contract owner check

Severity: Informational

In `contracts/staker/src/contract/exec.rs:104`, `183`, `939`, `1058`, and `1430`, there are comments stating that the code checks if the sender is the contract owner. However, these functions are not part of the `PROTOCOL_MODIFIERS`, meaning they can be executed by users with appropriate roles, not exclusively by the contract owner.

This discrepancy between the comments and the actual access control implementation can cause confusion for developers and auditors about who can execute these functions.

Recommendation

We recommend fixing the comments to accurately reflect that the functions check for users with appropriate roles or admin privileges rather than specifically checking for "contract owner".

Status: Resolved

32. Admin overwriting in ledger registration may cause confusion

Severity: Informational

In `contracts/staker/src/contract/exec.rs:193-196`, the `execute_register_ledger` function replaces the admin information from the `LedgerInstantiateMsg` struct, with the address of the staker contract.

However, the caller could have expected that the admin value would have been the one passed to the `instantiate_msg` in `RegisterLedger` message.

This creates potential confusion where callers may not realize their specified admin value will be overwritten, leading to unexpected behavior in ledger contract administration.

Recommendation

We recommend either only passing `LedgerSettings` to the `RegisterLedger` message, and completing with the admin information, or clarifying in documentation that the admin will be overwritten by the staker contract.

Status: Resolved

33. Misleading withdrawal request ID and inaccurate completion time

Severity: Informational

In `contracts/staker/src/contract/exec.rs:591`, the `withdrawal_request_id` is calculated as the count of `USER_WITHDRAWAL_REQUESTS`. However, there is no actual ID in `WithdrawalRequest`, and this count can be reduced during `process_resolvable_requests` in `Sync` and increased during `Redeem`. Calling it an ID and returning it as an attribute to `Redeem` is misleading, as it represents only the count of user withdrawal requests, not a unique identifier.

Additionally, in line `594`, the `completion_time` is calculated by adding 21 days to the current time using the magic number `21 * 24 * 60 * 60`. This is problematic because the 21-day period is not necessarily correct as it is a parameter in each ledger contract, the

unbonding does not start within `Redeem` but within `Sync` which can happen at any time later, and it uses magic numbers that should be avoided.

This misleading information can confuse users about the actual status and timing of their withdrawal requests.

Recommendation

We recommend renaming the variable `withdrawal_request_id` to `user_withdrawal_requests_count` and updating the corresponding attribute name as well.

We also recommend either completely removing `completion_time`, or if an indication of potential completion time is needed, using a configurable variable stored in the staker contract. If the latter option is chosen, we recommend removing the `unbonding_time` from the `LedgerInstantiateMsg ledger_settings` as it could be taken directly from the staker contract instead.

Status: Resolved

34. Inconsistency in paused contract checks may lead to confusion

Severity: Informational

In `contracts/staker/src/contract/exec.rs:441` and `524`, both functions `execute_deposit` and `execute_redeem` do not have the check for paused contract within their function code, unlike other functions. Instead, they have the checks within the main execution handler in `contracts/staker/src/contract.rs:138-141` and `145-148` respectively.

Additionally, some functions like

- `execute_update_admin`,
- `execute_reconcile_ledgers`,
- `execute_collect_unbonded_amounts`,
- `execute_update_fees_percentage`,
- `execute_update_treasury_address`,
- `execute_distribute_rewards_and_fees`,
- `execute_sync`,
- `execute_add_protocol_modifiers`,
- `execute_remove_protocol_modifiers`,
- `execute_add_role`,
- `execute_remove_role`,
- `execute_add_user`,
- `execute_remove_user`,

- `execute_distribute_losses`

execute regardless of whether the contract is paused, with no clarification on why some functions should continue operating during pause while others should not.

This inconsistency creates confusion about the intended behavior during contract pause and may lead to unexpected function availability or restrictions.

Recommendation

We recommend using a consistent approach for paused contract checks and providing clear documentation on why certain functions should not run while paused while others should remain operational during pause periods.

Status: Resolved

35. Unused variables and functions

Severity: Informational

In `contracts/staker/src/contract/exec.rs:1445-1446`, the variables `total_supply` and `funds_raised` are loaded but never used in the subsequent logic. Additionally, the `get_token_total_supply` function in lines `393-405` is only used in the call in line `1445`, making it effectively unused as well.

There are other unused variables throughout the codebase that contribute to code clutter and potential confusion. Running `cargo clippy` reveals numerous warnings and errors related to unused variables, dead code, and other linting issues that should be addressed.

Recommendation

We recommend removing the unused variable assignments in lines `1445-1446` and subsequently removing the `get_token_total_supply` function. For variables that must remain for future use, they should be prefixed with underscore to indicate intentional non-use. We also recommend resolving all warnings and errors from `cargo clippy` to maintain code quality and eliminate dead code throughout the codebase.

Status: Resolved

36. Incomplete documentation of default protocol modifiers

Severity: Informational

In `contracts/staker/README.md:442-456`, a list of functions are presented as protocol modifiers automatically added at instantiation.

However, the `default_protocol_modifiers` in `contracts/staker/src/contract/ init.rs:126-147` include additional functions:

- `execute_migrate_token`,
- `execute_migrate_ledger`,
- `execute_redelegate_ledger`,
- `execute_initiate_emergency`,
- `execute_withdraw_emergency_funds`,
- `execute_emergency_open_more_unbondings`

This creates incomplete documentation where some protocol modifiers are not listed, potentially causing confusion about which functions are automatically granted protocol modifier privileges during contract instantiation.

Recommendation

We recommend including all default protocol modifiers in the documentation to provide complete and accurate information about the contract's initialization behavior.

Status: Resolved

37. Inefficient filtering of available ledgers for unbonding

Severity: Informational

In `contracts/staker/src/contract/exec.rs:1338-1340`, `available_ledgers` is computed by filtering ledgers that have unbonding capacity, comparing the `unbonding_requests_count` with `max_unbonding_requests`. However, ledgers with 0 `bonded_amount` are still selected, although they do not have any unbonding capacity, causing unnecessary subsequent process browsing through ledgers that may not have unbonding capacity.

This results in inefficient processing where ledgers without bonded tokens are considered for unbonding operations, wasting computation cycles on ledgers that cannot actually perform unbonding.

Recommendation

We recommend filtering ledgers with positive `bonded_amount` for optimization to ensure only ledgers with actual unbonding capacity are considered in subsequent processing.

Status: Resolved

38. Redundant expensive `get_ledgers` call in `Sync` function

Severity: Informational

In `contracts/staker/src/contract/exec.rs:1472`, `all_ledgers` is initialized with a call to `get_ledgers`. Then, in line `1512`, `ledgers_for_bonding` is initialized with the same expensive call, with the comment "Get ledgers again for bonding (in case they were updated)".

However, the `get_ledger` information is retrieved from the ledger contract, which cannot have been modified during the course of a function in the staker contract.

This results in unnecessary gas consumption and processing overhead by performing the same expensive operation twice within a single function execution.

Recommendation

We recommend reusing the `all_ledgers` and not calling `get_ledgers` again to avoid redundant contract queries and reduce gas costs.

Status: Resolved

39. Incorrect documentation for withdraw completed unbonding function

Severity: Informational

In `contracts/staker/src/contract/exec.rs:1662-1686`, `rustdoc` comments mention that users need to provide the completed unbonding ID to withdraw, while the function actually withdraws for all completed unbondings of the user.

Additionally, the errors `CompletedUnbondingNotFound` and `UnauthorizedWithdrawal` cannot be returned by the function, while another error: `NoCompletedUnbondings` can, if there is no completed unbonding for the user.

This creates misleading documentation that does not accurately describe the function's behavior and lists incorrect error conditions, potentially confusing developers using this function.

Recommendation

We recommend updating the documentation to be consistent with the behavior of the function, correcting both the description of the withdrawal process and the list of possible errors.

Status: Resolved

40. Migration functions cannot pass custom parameters

Severity: Informational

In `contracts/staker/src/contract/exec.rs:1996-2030`, the `execute_migrate_token` function allows the staker contract admin to run a migration of the token contract. Similarly, in lines `2061-2109`, the `execute_migrate_ledger` function allows to run a migration of all the registered ledger contracts.

However, the code only allows migrating to a new code ID, but does not allow to pass custom parameters in the migration, as the only message passed is `MigrateMsg { new_admin: None }` or `LedgerMigrateMsg { new_admin: None }`.

This limits the flexibility of migrations, preventing the admin from passing necessary parameters that might be required for more complex migration scenarios involving data transformation or configuration updates.

Recommendation

We recommend adding a possibility for the staker contract admin to send generic parameters that would be passed to the migration handlers, for future-proof token or ledger migration process.

Status: Resolved

41. Redundant paused status checks

Severity: Informational

In `contracts/ledger/src/contract/exec.rs:380-384` and `486-490`, the contract checks for paused status within the `execute_reconcile` and `execute_redelegate` functions respectively. However, these checks are redundant as the same validation is already performed in the main execution handler in `contracts/ledger/src/contract.rs:53-56` and `60-63` before these functions are called.

The duplicate checks add unnecessary gas costs and code complexity without providing additional security benefits.

Recommendation

We recommend removing the redundant paused status checks from the individual execution functions since the validation is already handled at the entry point in the main contract execution handler.

Status: Resolved

42. Contracts should implement a two-step ownership transfer

Severity: Informational

The three contracts within the scope of this audit allow the current owner to execute a one-step ownership transfer. While this is common practice, it presents a risk for the ownership of the contract to become lost if the owner transfers ownership to an incorrect address.

A two-step ownership transfer will allow the current owner to propose a new owner, and then the account that is proposed as the new owner may call a function that will allow them to claim ownership and actually execute the config update.

This can be found in:

- Ledger contract: `contracts/ledger/src/contract/exec.rs:241-276`,
- Staker contract: `contracts/staker/src/contract/exec.rs:349-384`,
- Token contract: `contracts/token/src/contract/exec.rs:284-315`.

Recommendation

We recommend implementing a two-step ownership transfer. The flow can be as follows:

- The current owner proposes a new owner address that is validated and lowercased.
- The new owner account claims ownership, which applies the configuration changes.

Status: Resolved

43. Burn function address parameter is misleading

Severity: Informational

In `contracts/token/src/contract/exec.rs:135-165`, in `execute_burn` accepts an `address` parameter that is only used for event logging, while the actual burn happens on tokens sent by the contract owner.

The function requires the contract owner to send tokens via `must_pay(&info, &full_denom)?` at line 160, then burns those tokens. The `address` parameter is only used in the burn event but does not affect the actual burn operation.

A misleading interface could confuse integrators about the function's behavior. Only the contract owner can call this function.

Recommendation

We recommend clarifying the parameter's name, removing or documenting its purpose.

Status: Resolved

44. Confusing event logs

Severity: Informational

In `contracts/ledger/src/contract/exec.rs:314-320`, the `execute_update_ledger_settings` function preserves the validator address as immutable (line 314), but the event attributes suggest it can be changed by showing "old" and "new" validator addresses that are always the same

Event logs (lines 329-330) are confusing, suggesting the validator address was updated when it actually remains unchanged.

Recommendation

We recommend updating the event attributes to either omit the validator address entirely or use a single "validator" attribute instead of "old_validator" and "new_validator" to clarify that this field is immutable.

Status: Resolved

45. Debug code in production

Severity: Informational

In `contracts/ledger/src/contract/exec.rs:390-392`, and `packages/zmacros/src/lib.rs:74` contain debug code which can cause issues if accidentally enabled.

Recommendation

We recommend removing all debug artifacts, especially the commented panic and production `println!` statements.

Status: Resolved

46. Missing documentation for multiple functionalities

Severity: Informational

Multiple complex functions lack adequate documentation:

- `calculate_stzig_price` - no explanation of pricing formula.
- `find_suitable_ledgers_for_unbonding` - complex algorithm undocumented.
- `distribute_losses` - no explanation of loss distribution logic.

Recommendation

We recommend adding comprehensive inline documentation for the abovementioned functions.

Status: Resolved

47. Lack of event emissions prevents off-chain monitoring and integration

Severity: Informational

Critical state-changing operations across all contracts lack event emissions, preventing off-chain services from tracking protocol activity, debugging issues, or building integrations:

- In `contracts/staker/src/contract/exec.rs` in `execute_distribute_losses` - no event for loss distribution.
- In `contracts/staker/src/contract/exec.rs:765` in `execute_reconcile_ledgers` - no event for reconciliation.
- In `contracts/ledger/src/contract/exec.rs:369` in `execute_reconcile` - no event for slash detection.
- Migration functions - no events for admin changes.

Recommendation

We recommend implementing comprehensive event emissions using CosmWasm's `add_event` for all state changes, including amounts, addresses, timestamps, and operation types.

Status: Resolved